AFIT/GE/ENG/92D-25

A GENERIC TEMPLATE EXTRACTOR (GENTEX) IN C FOR VLSI DESIGN VERIFICATION

THESIS

Kenneth J. McClellan, Jr.
First Lieutenant, USAF

AFIT/GE/ENG/92D-25

Approved for public release; distribution unlimited

AFIT/GE/ENG/92D-25

A GENERIC TEMPLATE EXTRACTOR (GENTEX) IN C FOR VLSI DESIGN VERIFICATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Kenneth J. M°Clellan, Jr., B.S.E.E.

First Lieutenant, USAF

December 1992

Acknowledgements

I would like to express my sincerest appreciation to my wife, Linda. I could not help much with the chores around the house. I did not help much with our new daughter Kaila. I spent countless nights in the VLSI lab. Through all of this, she still supported me, loved me, and tried to help wherever she could.

I would also like to thank my thesis advisor, Major Kim Kanzaki, and my committee members, Lt Colonel William Hobart and Major Mark Mehalic, for their help and support. I would like to give a special thanks to recently separated Captain Keith Jones for his help with the programming and for helping me come up with a doable thesis in the middle of July.

Kenneth J. McClellan, Jr.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Explanation
AFIT	Air Force Institute of Technology
AU	Air University
CAD	Computer-Aided Design
CMOS	Complementary Metal-Oxide Semiconductor
EDIF	Electronic Design Interchange Format
GENTEX	GENeric Template EXtractor
GES	Generalized Extraction System
GND	Ground or Zero Volts
MOS	Metal-Oxide Semiconductor
SGE	Simulation Graphics Environment
STOVE	Sim TO VHDL Extraction
t-gate	Transmission Gate
Vdd	Supplied Voltage, High, or +5 Volts
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large-Scale Integration

Abstract

The problem of VLSI design verification through circuit extraction was analyzed. The problems of creating a simple template format, the permutability of pins, maintaining connectivity, and performance were focused on. A generic template extractor (GENTEX) was developed in the C programming language for use as a testbed to find solutions to these problems. Six different extraction algorithms were tested with GENTEX and compared based on performance. EDIF translation programs were used to interface with GENTEX on both the input and output sides. One translation program converted an EDIF representation of a schematic into the template format used by GENTEX. The other translation program converted the output of GENTEX into a schematic in EDIF. The results of the performance analysis showed that an extraction algorithm based on searching the data structures by node rather than by component type provided the best performance. The results also showed that comparing the number of connections to a node within a template to the actual number of connections to a node within the circuit being extracted, not only eliminated any connectivity problems but also increased performance.

A GENERIC TEMPLATE EXTRACTOR (GENTEX) IN C FOR VLSI DESIGN VERIFICATION

I. Introduction

1.1 Background

VLSI Design Environment. The Very Large-Scale Integrated (VLSI) circuit design process may vary slightly between design environments. However, the overall process is basically the same. Figure 1 shows the VLSI design process at the Air Force Institute of Technology (AFIT).

Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is used to simulate the design before the process of realizing it in silicon begins. VHDL allows the designer to detect potential problems, try different combinations or configurations, and optimize the design before the transistor layout is done in Magic.

The layout in Magic represents the physical layout of the final circuit. The relative size and position of the n-type material, p-type material, polysilicon, and metal depicted graphically in Magic is true in the final product. Therefore, it is important to be sure the layout is correct. This makes testing of the layout extremely important.

MEXTRA translates the mask layout representation into a transistor net list representation. In this process, all hierarchical information is lost. The result is a list of

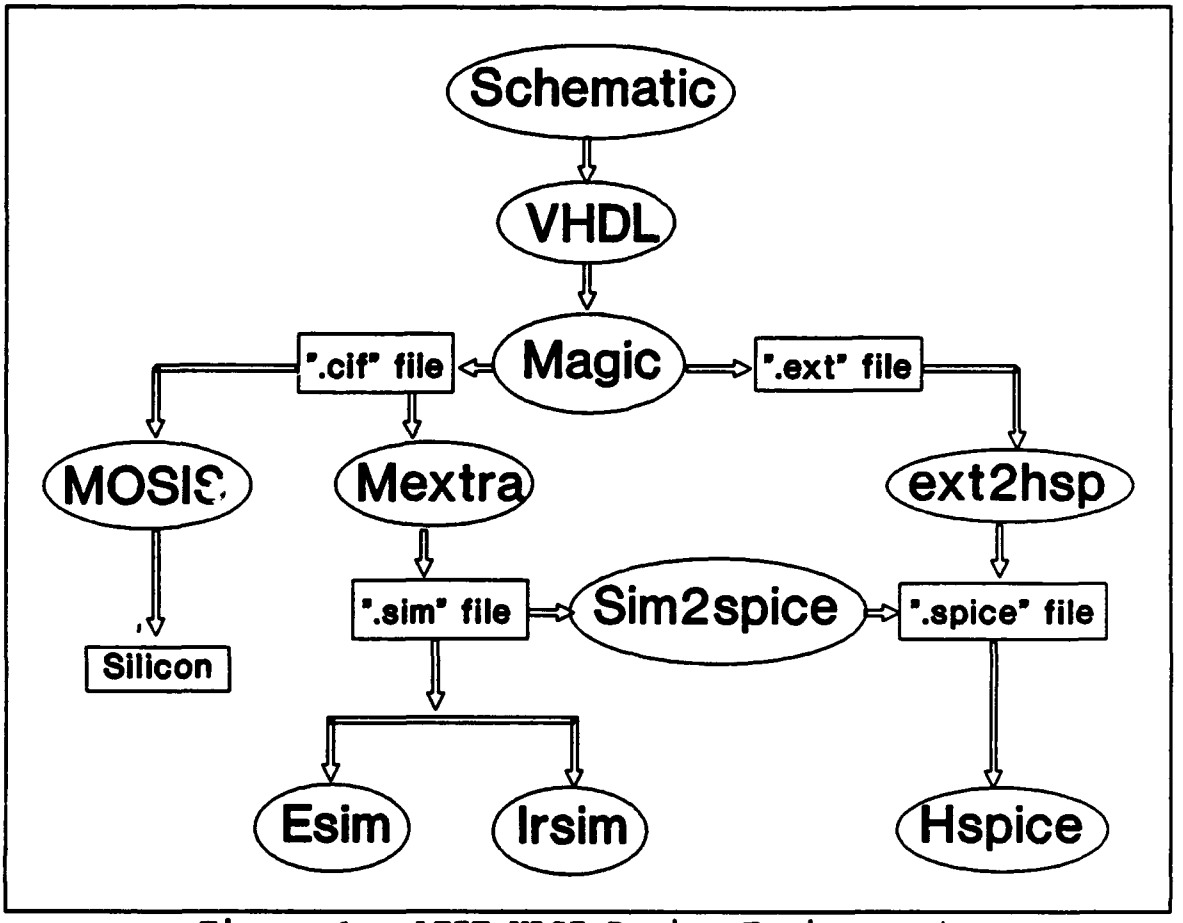


Figure 1. AFIT VLSI Design Environment.

transistors with the node names of their pins, the capacitances associated with certain nodes, and the resistances associated with certain nodes. This transistor net list is given a ".sim" extension.

The transistor net list may then be used as the input to ESIM or IRSIM. These programs are switch-level simulators. Logic information of the circuit is determined, but little or no timing information is produced.

HSPICE is used to simulate the circuit in steady-state, transient, and frequency domains [1]. Either the ".ext" file produced from Magic or the ".sim" file produced from

MEXTRA may be converted to the HSPICE format via EXT2HSP or SIM2SPICE, respectively.

MOSIS is a metal-oxide semiconductor fabrication service. The ".cif" produced by Magic is used by MOSIS as the mask description. MOSIS returns the circuit that was specified in the ".cif" file realized in silicon as an integrated circuit chip.

Testing. Current VLSI design environments lack sufficient testing tools and the existing testing tools, lack standard interfacing capabilities. These inadequacies cost time, money, and quality of the final product.

The earlier in the design process that an error is discovered, the less costly it is to correct it. The costs to correct an error include the time and money to change the design in the CAD tools and to redo the testing. Unfortunately, errors are often realized in silicon due to the lack of testing before fabrication. Some of these errors may even make it to the final product. These errors may result in less efficient operation, inaccurate results, or even system failure. Even if an error is detected after it has been realized in hardware, due to time and cost constraints, it may not be totally corrected or it may even be ignored. Thus, the quality of the final product suffers from failure to detect problems early in the design process.

Most of the CAD tools used in VLSI design have some form of testing incorporated into the system. Generally,

the testing done in CAD tools is based on a model of the design, and therefore, is not necessarily indicative of proper operation in hardware. The results only indicate if some errors for which the tool specifically checks were detected. Since it is impossible to write a program to detect every error, it is important to use several different testing tools which may be better at detecting different types of errors. As stated above, the earlier an error is detected, the less costly it is to correct it.

There are several different techniques and approaches to testing VLSI designs. The approaches may generally be classified as validation or verification [2]. Validation is the process of demonstrating that the responses of the circuit match the responses desired in the design. Verification is the process of demonstrating that the components of the circuit match the components of the design.

Validation. Validation is currently the method of testing used in most VLSI design environments. However, validation of VLSI designs is quickly becoming intractable due to the high number of transistors in current designs. Validation generally refers to simulation. Simulation has a problem with exhaustively testing the circuit. Only simple circuits may be exhaustively tested. For example, a 32-bit adder/subtractor has 66 inputs: two 32-bit operands, a carry in, and a function bit. This means that there are 2^{66} input

combinations to test. Assuming a simulator can completely simulate 1024 input vectors per second, it would take over two billion years to exhaustively test a 32-bit adder/subtractor. As a result, subsets of the possible input and output vectors must be used. This decreases the level of confidence in the correctness of the design. Choosing the set of input and desired output vectors to be tested also becomes a problem [3]. Even if an exhaustive sequence could feasibly be used to test a circuit, it is not guaranteed that all flaws would be found [2].

Switch-level simulators are generally used to validate mask layout descriptions [4] and logic designs. There are some problems with switch-level simulators. They are generally based on steady-state models [2]. If a circuit is not designed to be in steady state conditions such as an oscillator, certain sequential circuits that produce race conditions, or circuits with feedback loops, the switch-level simulator will not work properly, if at all.

Event-driven simulators are generally used at the gate level of digital circuits [2]. Event-driven simulators do not have the problems that switch-level simulators do with non-steady-state circuitry. However, the problems with the number of test cases still exists.

Verification. VLSI verification is in several ways a better approach than validation. As opposed to validation, verification can show the correctness of a

design for all possible inputs and outputs [5]. If the design can be verified to a certain level of abstraction, it may then be simulated at that higher level of abstraction. This decreases the required simulation time to validate the circuit.

Verification is also a valid means to help document components whose documentation is either partially or totally missing. The Department of Defense (DoD) has had major problems replacing such components [2]. A standard library of components may be used as templates to be found in the unknown circuit. This allows the portions of the unknown circuit that contains standard cells to be extracted hierarchically and documented with documentation consistent with the standard cells.

1.2 Problem

There were no efficient methods for verifying VLSI circuits. Therefore, there was a need for a circuit extraction program to verify transistor layouts that used a simple template format and had high performance.

1.3 Scope

The scope of this thesis was limited to the VLSI design environment at AFIT. The EDIF used was EDIF 2 0 0 level 0 and was tailored to the CAD tools at AFIT. EDIF was used because it is a national standard for design interchange, and therefore, should be compatible with other software

packages. Test circuits were also limited to those available at AFIT that had schematics.

Several assumptions were made through the course of this thesis. It was assumed that the user would provide the ".sim" file and either the templates or the schematics of the templates to be extracted. This thesis also assumed that the circuits being examined are complementary metal-oxide semiconductors (CMOS) technology. In theory the techniques and approaches used in this thesis should work on any VLSI technology, but because the AFIT environment is almost exclusively CMOS, so was the focus of this thesis.

1.4 Approach

The approach of this thesis was to verify transistor layouts by demonstrating that the transistors do indeed comprise the desired higher-level circuit. This was done by bridging the gap between the schematic capture process and the ".sim" file depicted in Figure 1.

Schematics of particular components were converted to EDIF. As part of this thesis, a conversion utility was created to convert from EDIF to the template format used by the extraction program. In this manner, a hierarchy of templates was created to be used to extract the circuit from the transistor net list.

A program called GENeric Template EXtractor (GENTEX) was coded in the standard C programming language. This program extracts higher-level components from a net list

based on templates provided by the user. The net list generally starts as a transistor net list provided by MEXTRA as the ".sim" file. As GENTEX extracts components, the transistor net list is transformed to a component net list that meets the input requirements of GENTEX. This allows intermediate files to be made for later analysis. The intermediate files allow incorrect templates to be corrected without re-extracting all previous work. Intermediate files also allow circuits to be documented (with the component net list and the corresponding schematic) at different levels of hierarchy.

After the circuit was extracted, either partially or totally, another conversion utility created as part of this thesis was used to convert the component net list to EDIF. The EDIF file was then be imported into the Simulation Graphics Environment (SGE) of Synopsys or in CAPFAST of Three Phase Logic as a schematic. The relative positions of the components and the interconnectivity of the components were displayed as they existed in the transistor layout of the circuit.

There are several advantages to having a closed-loop system. The first is to allow as much automation of the process as possible. This saves valuable man-hours. Other advantages are derived from the fact that a schematic of the transistor layout was created at a higher level of hierarchy. One such advantage is that compliance between

actual layout and desired layout can quickly be determined either visually at high levels of abstraction or through simulation at lower levels of abstraction. Another such advantage is that if the actual layout varies from the predicted layout of the circuit, or if the layout of the circuit is originally unknown, the schematic can be used to generate VHDL and be simulated.

The method of extraction by GENTEX was also examined. Different approaches to the pattern matching within the extractor were compared. The approaches were compared and evaluated based on performance and accuracy.

1.5 Materials and Equipment

Several software packages and a computer systems were required to complete this thesis. The VLSI laboratory at AFIT provided the necessary software running on a network of mostly Sun SPARC II work stations running SunOS Release 4.1.2. CAPFAST 2.2 of Phase Three Logic and SGE of Synopsys 2.2 were the schematic capture tools used. Synopsys provided the VHDL simulator and design analyzer to produce EDIF 2 0 0 files from the VHDL. CAPFAST was capable of reading or writing EDIF. Magic 6.3 and MEXTRA were also present on the system. They were needed to produce the ".sim" files, although most of the ".sim" files were provided by other projects. Finally, a standard C compiler was provided on the Suns.

1.6 Sequence of Presentation

Chapter 2 is a literature review of the current status of VLSI verification through extraction. Current and past approaches to extraction and their respective qualities are discussed.

Chapter 3 discusses the problems previous programs encountered, the methodology used in this thesis, and the six extraction algorithms tested in this thesis. The specifics and justification of the approaches of this thesis are presented.

Chapter 4 is a description of GENTEX. This chapter discusses how GENTEX works and how it tries to avoid some of the problems encountered with other extraction systems.

Chapter 5 presents and analyzes the results and findings of the different extraction methods examined in this thesis. The respective performance measurements of the six extraction algorithms are compiled, analyzed, and displayed in this chapter.

Chapter 6 discusses the modifications made to the extraction programs tested in Chapter 5 and the results of those modifications. In particular, the modifications to improve memory usage and performance are discussed.

Chapter 7 presents and analyzes the results and findings as a result of the EDIF interfaces. The problems encountered with EDIF itself and the EDIF interface with other systems are discussed and analyzed.

Chapter 8 draws conclusions based on the results of the previous three chapters, summarizes the thesis, and gives recommendations based on this thesis.

Appendix A is a user's manual for the verification system developed in this thesis. Appendix A is designed to be a stand-alone document.

Appendix B contains the program code of the programs developed in this thesis. The program code for EDIF2TMP, GEN2EDIF, the extraction programs, and the modified program discussed in Chapter 7 is contained in this appendix.

II. Review of Current Extraction Programs

2.1 Introduction

VLSI design verification through circuit extraction has been the topic of several studies. This chapter describes the approaches used and the results of these studies. Particular attention is paid to Sim-TO-VHDL-Extractor (STOVE) and the Generalized Extraction System (GES). Both were recently completed extraction systems at AFIT.

In most of the earlier efforts, circuit extraction was viewed as simply matching templates of subcomponents within a circuit and replacing the subcomponents with a single component. However, this approach was characterized by problems maintaining connectivity and a limit of less than 10,000 transistors [2].

With the evolution of artificial intelligence, the approach to verification changed [6]. Most of the recent circuit extraction systems were created with an artificial intelligence programming environment, an expert system.

2.2 STOVE

STOVE was started as part of a thesis project at AFIT by D. Gallagher and was later modified and completed by Mark Petre of Systems Research Labs (SRL) through a contract between AFIT and SRL [7]. STOVE was written in the C programming language and extracts from a ".sim" file to VHDL

code. STOVE extracts twenty-four components using subroutines coded in standard C.

STOVE is capable of extracting two levels of hierarchy [7][8]. The input file is considered to be the transistor level. The first extraction level is called the gate level. The gate level consists of inverters, t-gates, and clocked inverters [7][8]. Any transistors not used to make these structures are copied to this level.

The second and final level of extraction of STOVE is called the logic level. Structures such as AND gates and resettable master/slave flip-flops are recognized and extracted at this level [7]. Any transistors or gates from the previous level of extraction not used to make the structures at this level are copied to this level.

More components may be added to the search by changing the source code and recompiling the program [7]. A subroutine in C must be written as well as slightly modifying parts of the original code. It is possible that the existing data structures will not support the added component, in which case a new data structure and an additional subroutine are required [7].

2.3 GES

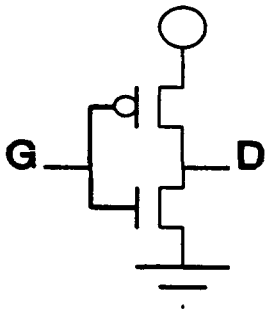
GES was started by M. Dukes as a master's degree thesis, and was later significantly upgraded as part of his dissertation. GES was originally meant to be a Prolog version of STOVE that would fix some of the problems found

with STOVE, thus the original name STOVE_P [8]. The name was changed from STOVE_P to GES during the dissertation work of Dukes.

STOVE_P, like the C version of STOVE or STOVE_C, starts with a ".sim" file produced by MEXTRA or EXT2SIM and ends with a VHDL description of the circuit. The ".sim" file first passes through a translator called SIM2PRO that converts the ".sim" file into the proper Prolog format [8]. The Prolog extraction routine called TRANS is then used to iteratively perform the extractions on the transistor and component databases [8]. Finally, the Prolog output is converted to VHDL through another translator called PRO2VHDL [8].

In the conversion from STOVE_P to GES there were many additions and upgrades. GES now performs five main tasks: 1) formal hardware verification, 2) reverse engineering of undocumented designs, 3) detection and location of errors in a design, 4) assistance in incremental documented design, and 5) critical path analysis [2]. The first four tasks are variations of applications of circuit extraction. The fifth task, however, provides timing information of pin-to-pin critical paths based on the resistances and capacitances provided by MEXTRA [2].

GES, like STOVE, also allows new components to be added to the extraction routine. Like STOVE, GES comes with a standard set of components that are recognized and extracted




```
inv:-
  ptrans(G,D,vdd),
  ntrans(G,gnd,D),
  remove_p(G,D,vdd),
  remove_n(G,gnd,D),
  asserta(inv(GD,1),
  fail.
inv.
```

Figure 2. CMOS Inverter Circuit with Prolog Description [9].

like inverters and transmission gates (t-gates). New components may be added by adding templates to the program. There are two ways to add templates to GES. The first is to make the template by hand. The template is actually Prolog code and an example is shown in Figure 2 of an inverter and the corresponding Prolog template. The other method to add templates to GES is to allow GES to generate the templates from a structural description in hierarchical VHDL.

2.4 Other Related Verification Tools

There certainly have been other efforts besides the two mentioned above. These other efforts vary widely in their approach to the problem of VLSI verification. Most efforts use an expert system to solve the problem; however, Pascal and C are also used.

The majority of the previous efforts apply a backward-chaining expert system such as Prolog. VERify CONnectivity (VERCON) [10] uses Prolog and an extraction process very similar to GES; however, VERCON produces the templates used

in the extraction from the hierarchical information contained in the ".ext" file generated by Magic. VERification Assistant (VERA) [11] and WOMBAT [12] use a library of templates created by the user to extract hierarchical components. VERIFY also uses Prolog, but with a different approach. VERIFY uses the results of a simulator to extract a boolean equation of the circuit [5]. Although this approach is called verification, it more closely resembles the definition of validation used in this thesis.

Although efforts that used forward-chaining expert systems for VLSI verification are rare, they do exist. S. Yarost wrote an extraction tool in CLIPS for his master's degree thesis [13] which is similar to an OPS5 [12] implementation of logic extraction. His program also uses templates to extract hierarchically. The templates are assumed to be provided by the user or in the current library. The circuit is verified by manually checking a graphical representation of the extracted circuit. The graphical representation shows the relative positions of the components as they existed in the Magic layout. The components have the name of the network, or node, to which they are attached adjacent to the pins of the component. No wiring is shown [13].

There are also several efforts that do not use an expert system. REX is a program that uses a complicated

algorithm dealing with "subgraph isomorphisms", similar to a hashing function, to extract hierarchical components of the circuit [14]. Like STOVE and GES, REX extracts only exact matches to the templates. Potential matches must be checked to confirm the match because the subgraph isomorphism technique does not guarantee an exact match. REX requires the input circuit to be in EDIF as well as the templates. The output may be either EDIF or KARL-III [14]. LOGEX, a Pascal program, also uses a complicated technique to extract a circuit [15]. LOGEX transforms the transistor view of the circuit into a node view, combines similar transistor branches hierarchically, and then determines a logical equivalent of the circuit including some timing information. The result is a simplified, rather than true representation of the circuit [15].

Finally, there are several extraction programs that verify the circuit at the transistor level without any hierarchical extraction. These programs generally determine the interconnections of the transistors and provide a schematic at the transistor level [16]. The schematic is then simulated.

2.5 Summary

There have been many efforts involving VLSI verification by proving the correctness of the transistor layout. Most of these systems use an expert system to hierarchically extract matches to templates such as GES uses

Prolog. Other languages are also used to program systems to hierarchically extract components such as in STOVE and REX. Some programs take the approach of determining the boolean or logical equivalents to the circuit instead of an exact match such as LOGEX and VERIFY. Finally, some limited timing information is also provided from some programs such as in GES and LOGEX.

III. Methodology

3.1 Introduction

This chapter discusses the problems encountered with the previous extraction programs discussed in Chapter 2. Potential solutions to these problems are addressed in terms of the development of GENTEX. The specifics of the six extraction algorithms are discussed. The two approaches to solving the permutability problem are examined. The data collection method and the program used to time the extractions are discussed. Finally, why EDIF was chosen to be the format for interfacing GENTEX with other programs is discussed.

3.2 Problems Previously Encountered

There were several problems with the extraction programs discussed in Chapter 2. These problems, addressed by GENTEX, may be combined into the following categories: speed, memory usage, accuracy, and ability to add new components.

Speed. The speed of most of the systems previously developed is less than desirable. The size of the circuits extracted is limited by the time it takes to extract them. For example, WOMBAT is time-limited to about 2,000 transistors [12]. Speed is an inherent problem with expert systems, such as Prolog. VERCON is admittedly slower than

other systems due to the use of Prolog [10]. Finally, forward-chaining expert systems such as CLIPS suffer further because the rule-firing order cannot be controlled and therefore the performance cannot be fine tuned [2].

Memory Usage. The amount of memory used by many of the previous systems is also a problem. Expert systems, in particular, generally use large amounts of memory. Programs that use an expert system like Prolog or CLIPS generally use so much memory that only small circuits can be extracted [12]. Yarost showed in his CLIPS implementation of an extraction program that memory usage goes up exponentially with the number of matches required within a template [13]. The CLIPS system exceeds the 100 megabytes of available memory when trying to extract ten or more templates from a 2,000 transistor file [2][13].

Accuracy. Some of the programs discussed in Chapter 2 have problems with maintaining connectivity. Figure 3 shows a NAND gate and a circuit that resembles a NAND gate. Notice that if the second circuit is extracted to a NAND gate, the connection of the remaining transistor to node "X" is lost. This loss of connectivity is encountered in previous systems due to programming error rather than limitations of the programming language. This is a problem anytime the user is required to write code for a template. GES addresses this problem and provides a means for the programmer to avoid it [2]; however, it is ignored in other

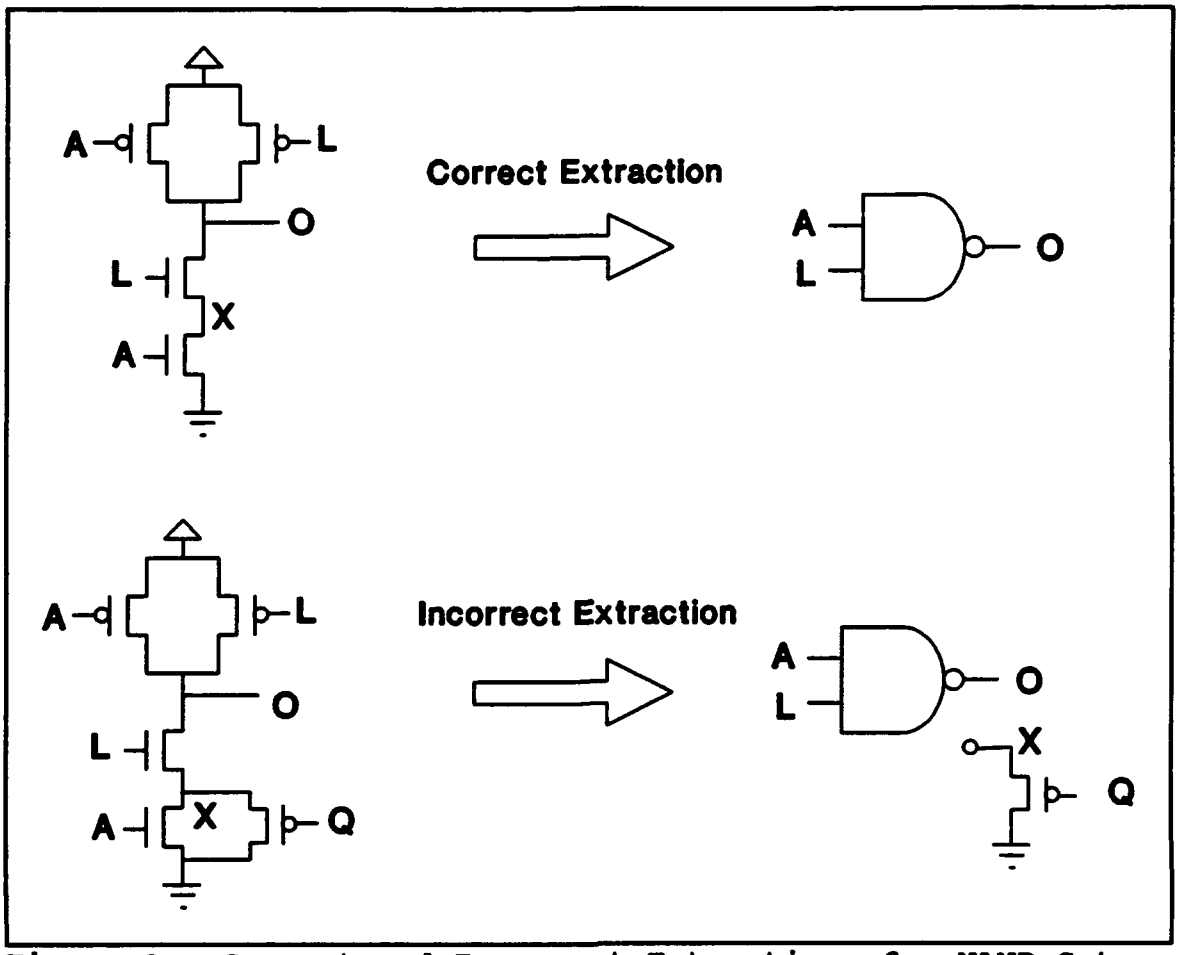


Figure 3. Correct and Incorrect Extraction of a NAND Gate.

programs such as STOVE and the CLIPS extraction program [2].

Many of the programs do not address the permutability of pins of a component, such as the interchangeability of two inputs of a NAND gate. GES addresses this problem only at the transistor level [2] and in NAND gates through the template. STOVE requires the user to write permutability into each routine to extract a new component. Permutability is not addressed by most of the other previous extraction efforts.

Other problems have occurred with accuracy besides the

loss of connectivity. Programs such as VERIFY [5] and LOGEX [15] do not extract the actual circuit. Instead, they extract boolean and logical equivalents of the circuit which may not be sufficient for verification of the design.

Finally, accuracy may be hindered due to the limitations within the programs. STOVE does not support a component that has more than two input or output pins [7]. The CLIPS program does not support any components with more than three permutable pins [13].

Ability to Add New Components. It is difficult to add new components to most of the previous systems. In order to add a new component to many of the systems, such as STOVE and GES, actual code must be written. GES considers a new component to be a template and attempts to solve the problem of adding new components with these templates. However, these templates are nothing more than Prolog code in the form of subroutine calls. In both STOVE and GES, the complexity of the templates increases non-linearly with the number of required matches within the template. Figure 4 shows the GES template for the NAND gate in Figure 3. Compare this with the GES template for an inverter in Figure 2. The code to find NAND gates in STOVE is approximately two pages long. The template for the NAND gate for GENTEX is in Figure 13 and will be fully explained in Section 4.4. Templates such as those in GES and STOVE require not only knowledge of the programming language, but also of the

```
nand :-
   ntrans(Agnd,X),
   ptrans(Avdd,O),
   not(gnd=O),
   not(vdd=O),
   more_nand(L,X,O,gnd),
   remove_p(A,vdd,O),
   remove_n(A,gnd,X),
   asserta(nand([A|L],O,1)),
   fail,
nand.

more_nand([A|L],X,O,P) :-
   ntrans(A,X,Y),
   ptrans(A,vdd,O),
   not(X=Y),not(Y=P),
   more_nand(L,Y,O,X),
   remove_n(A,X,Y),
   remove_p(A,vdd,O).

more_nand([A],X,O,_) :-
   ntrans(A,X,O),
   ptrans(A,vdd,O),
   not(X=O),
   remove_n(A,X,O),
   remove_p(A,vdd,O).
```

Figure 4. NAND Gate Template in GES [9].

internal workings of the programs themselves, making the templates prone to errors and difficult to debug.

VERA has a different approach to adding components to the extraction list. A large amount of complex data is required to add a component. The complexity of the data provides faster extractions, but the time required to produce the data more than nullifies the gain in performance [11].

3.3 Extraction Approach

There were several possible choices for the approach to

logic extraction. Previous systems use block extraction, boolean extraction, and logic extraction. These approaches have their individual advantages and disadvantages.

Block extraction is the replacing of one or subcomponents with a single higher-level component. The disadvantage of this approach is that it requires a template or other description of each new component in terms of its lower-level components [14]. The advantages of this approach are that the extracted circuit is an exact match of the original circuit, and it is technology independent [14].

Boolean extraction is the determination of the boolean equation of the circuit based on the results of simulations. The disadvantage of this approach is that the circuit must be simulated. Simulation is a lengthy process and requires considerable effort to determine the input vectors and desired output vectors. The advantage of this approach is that no templates are required to determine the boolean equations.

Logic extraction is the determination of an equivalent circuit to the circuit under test based on the interconnection of the transistors. The equivalent circuit is made up of standard components such as logic gates, registers, and buffers. Higher-level components are determined based on the recognition of repetitive sub-circuits. These sub-circuits are then represented as a single higher-level component with an associated logic

equation. The disadvantages of this approach are that it is much more complicated than the other two approaches. Some custom circuits may not be recognized, and the higher-level components must be hierarchically re-created because they are not standard components. The advantages of this approach are that no templates are required to do the extraction, and the extracted circuit is an exact match to the original.

The approach chosen in this thesis is block extraction. The advantages of block extraction stated above are part of the reason for this decision. The main reason for using this approach is that GENTEX already used the block extraction approach.

3.4 Extraction Algorithms

There are three different techniques used in matching the templates in the extraction process. Two different variations are used for each of the three techniques for a total of six extraction algorithms to test. GENTEX was modified so that there are six versions of the program that implement the six extraction algorithms.

The difference in the six programs is solely in the extraction algorithms. All six of the extraction algorithms match the templates subcomponent by subcomponent. The difference is that all six algorithms use their own approaches to find the subcomponents. As each subcomponent

is found, the variable node names are assigned to the actual nodes connected to the respective pins of the subcomponent.

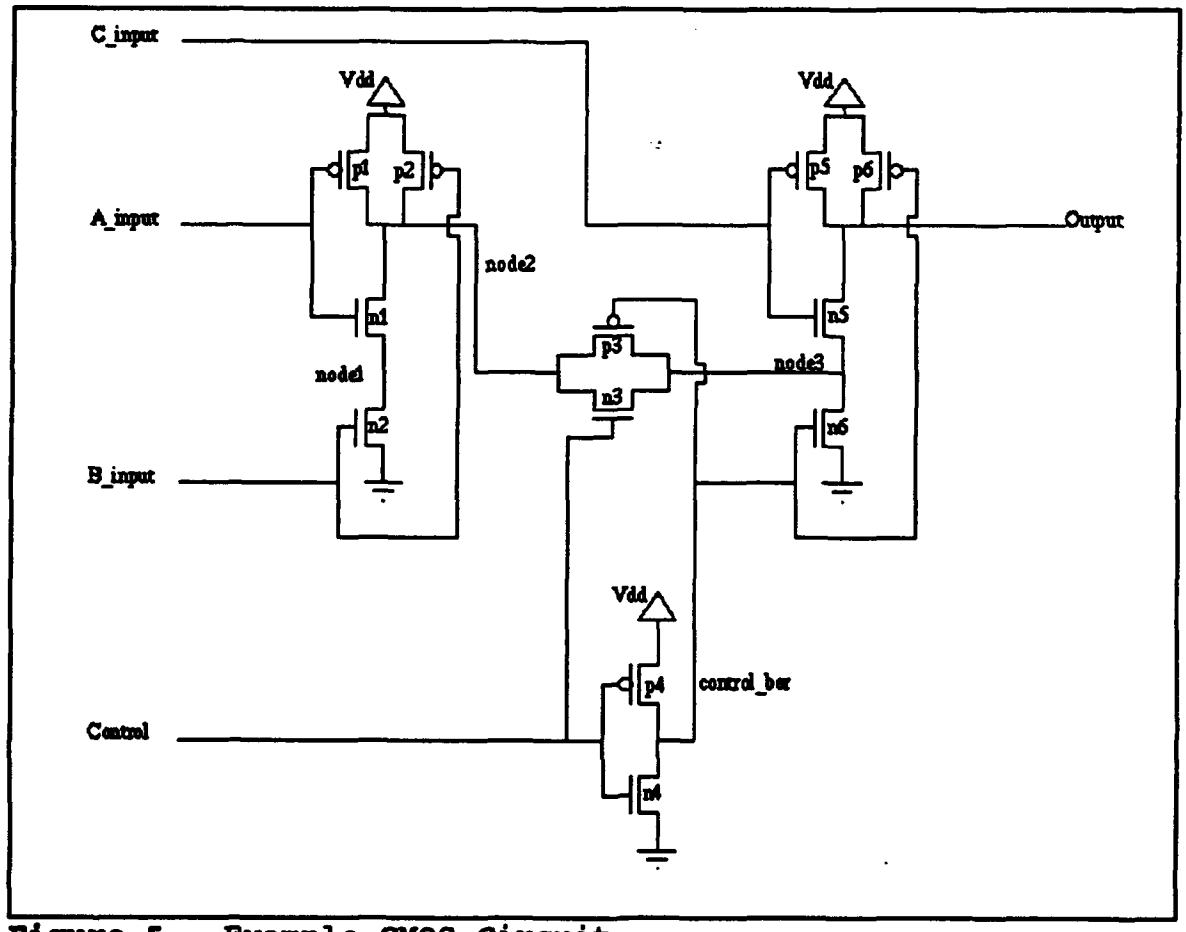


Figure 5. Example CMOS Circuit.

Figure 5[1] is a good example circuit for describing the extraction routines. This circuit contains one NAND gate, one t-gate, one inverter, and the remaining transistors resemble a NAND gate, but they do not comprise one. Figure 6 shows a NAND gate and its corresponding template. This is also used in the discussion of the six extraction algorithms.

[1] Note: This circuit is for illustration purposes only. It has no known real world application.

```
begin template;
   NAND VA VB Vout;
   p VA LVdd:1 Vout:1;
   p VB LVdd:1 Vout:1;
   n VA Vout:1 VN1:1;
   n VB VN1:1 LGND:1;
end template;
```

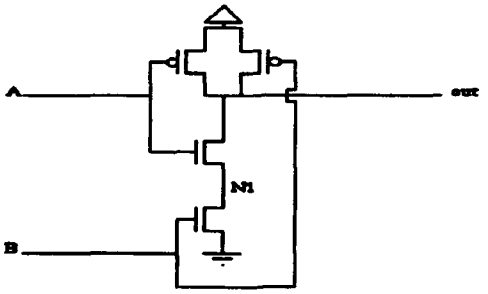


Figure 6. NAND Gate Template and Circuit.

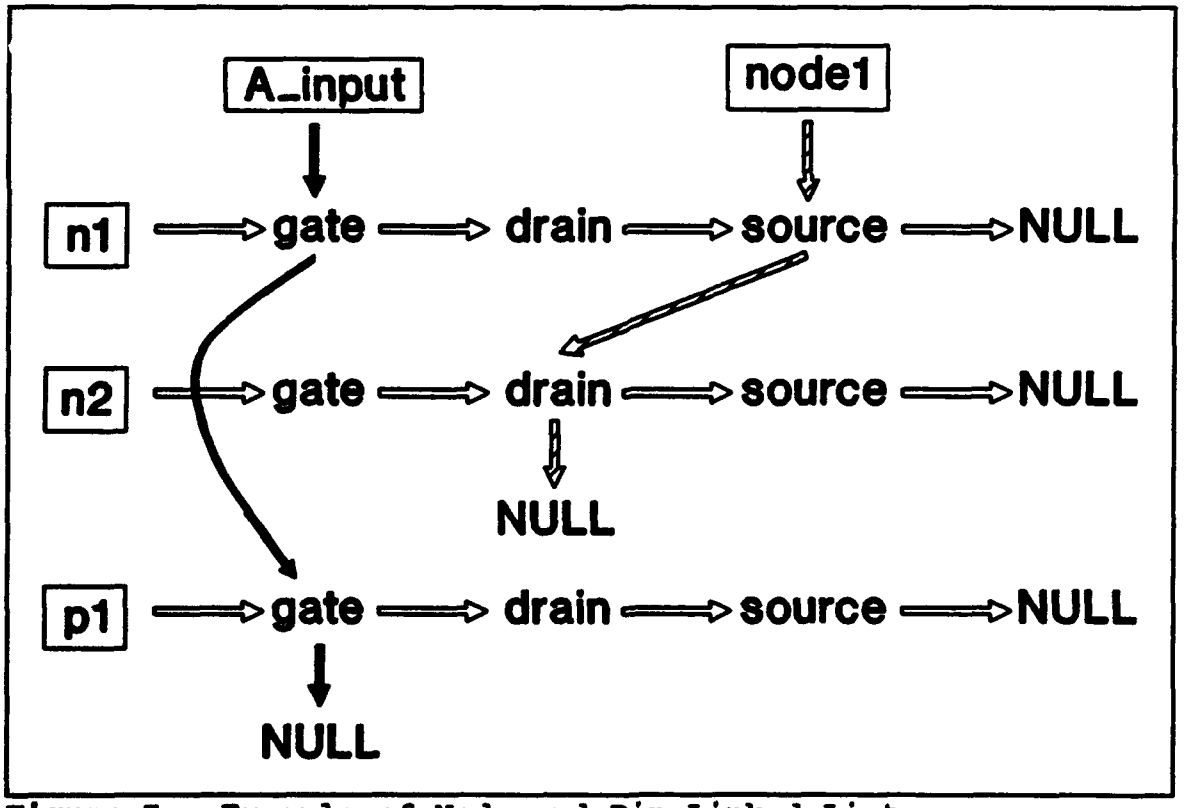


Figure 7. Example of Node and Pin Linked Lists.

Figures 7 and 8 show examples of how the data structures of the nodes, components, and pins are

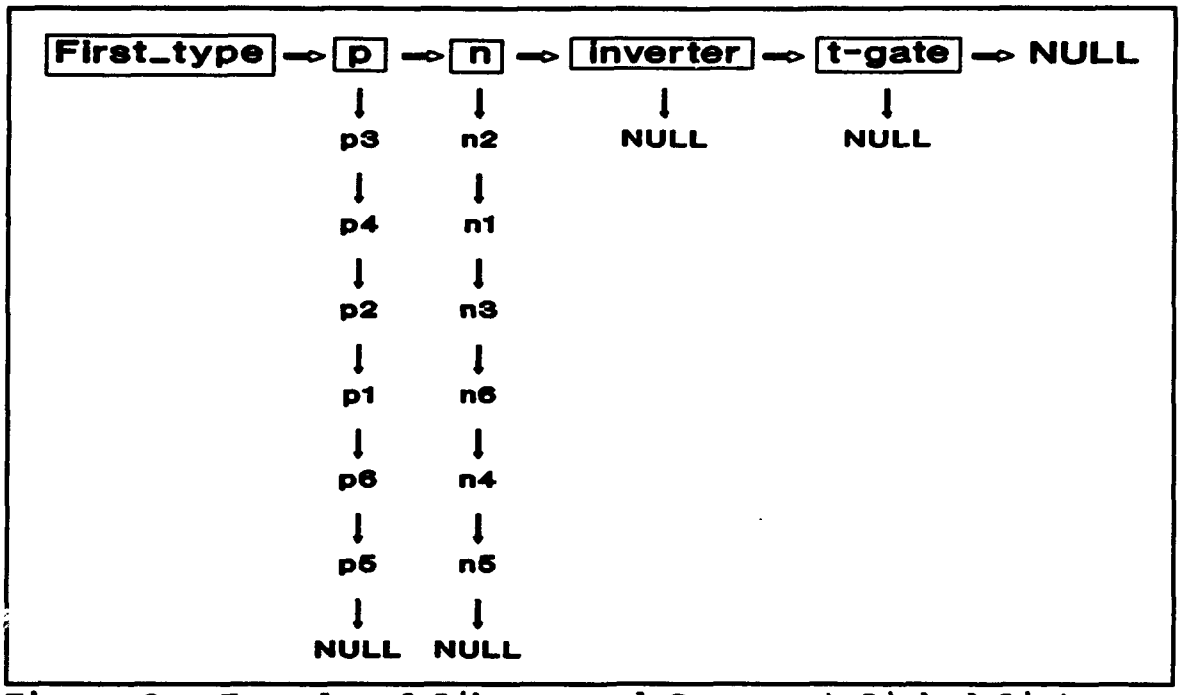


Figure 8. Example of Library and Component Linked Lists.

intertwined using the data from Figure 5. The difference in the three extraction techniques are in the way these data structures are used to find matches to the templates.

Figure 7 shows examples of node and pin linked lists. From Figure 5, the node "A_input" is connected to the gates of "n1" and "p1." Figure 7 shows the node "A_input" pointing to the gate of "n1," which points to the gate of "p1." Link lists, such as this one, allow nodes to be searched with very little overhead.

The other type of linked list in Figure 5 is called a pin linked list. These link lists connect the components with the data structures of their associated pins. The components in Figure 7 all have three pins, because they are transistors which have three pins. The length of these

linked lists depends on the number of pins of the components.

Figure 8 shows the component and library linked lists using the data in Figure 5. The horizontal linked list is the library linked list. The vertical linked lists are the component linked lists. The library linked list contains the component types. The component linked list contains all the components of a particular type. For example, the component linked list starting with the "p" in Figure 8 contains all the p-type transistors in the circuit in Figure 5.

The first technique used looks for the next subcomponent on a specific node. When the search for a subcomponent begins, the names assigned to each pin are checked. If the subcomponent contains any variable node names that have already been mapped to a specific node or if it contains any literal node names, those nodes are searched for the subcomponent that satisfies the requirements in the template. For example, the template in Figure 6 starts with a p-type transistor. This extraction technique would examine the node names of that subcomponent to determine if any of them are known. In this case, "Vdd" is known because it is a literal node name. The node with the name "Vdd" would be the only node searched for a p-type transistor. Any of the p-type transistors that are connected to "Vdd" in Figure 5 would match this subcomponent of a NAND gate. The

transistor chosen may be any of the five transistors connected to "Vdd". The transistor "p4" might be chosen, in which case the variable node name "A" would be mapped to the "Control" node and "out" would be mapped to "control_bar." The next p-type transistor ("p VB LVdd:1 Vout:1;") would then be searched for, because it is next in the template. In this case, both "Vdd" and "out" are known nodes because "out" was previously mapped to "control_bar." Either "Vdd" or "control_bar" would be searched for a match to the p-type transistor. In this case, there are no p-type transistors that have the proper connections to both "control_bar" and "Vdd." So no match is found, and the search for the previous subcomponent (the first p-type transistor) is restarted from the point where it was left off. Eventually, all NAND gates are found and extracted, in this case there is only one. Since this technique was the one originally programmed it is called GENTEX, and is referred to by that name for the remainder of this thesis.

The second technique looks for the next subcomponent, component by component. When the program is attempting to find a match for the subcomponent in the template, it searches only components of the same type as the subcomponent in the template. For example, in order to find the first p-type transistor of the NAND template in the circuit in Figure 5, every p-type transistor is checked until one is found that has the proper pin connections.

Transistor "p3" may be first in the linked list of p-type transistors (see Figure 8). In this case, "p3" does not have the proper pin connections because "Vdd" is not connected to it. The next p-type transistor in the linked list may be "p4." Again, "p4" matches the restrictions of the search and "A" is mapped to "Control" and "out" is mapped to "control_bar." The search for the next p-type transistor of the NAND template starts at the beginning of the linked list of p-type transistors again. Transistor "p3" is first and does not match. Transistor "p4" is next but has already been used. This technique searches the entire linked list for a match to the second p-type transistor in the NAND template. None is found and the search for the first p-type transistor resumes. For the reason that this technique searches component by component, it is referred to as GENTEXC.

The third and final technique uses a foreknowledge of an output pin of the template. The output pins of all complementary gates are on a node that has both a p-type transistor drain and an n-type transistor drain connected to it. In this extraction technique, all nodes that meet this

Figure 9. Example of an Output-Node Linked List.

criteria are marked and linked together as in Figure 9. The

last pin of the new component of a template is assumed to be the output pin. When the search attempts to match a template, the first subcomponent searched for is the first one listed in the template that is connected to the output node. For example, the node of the last pin in the NAND template is "Vout" in Figure 6. This is assumed to be the output node of the NAND gate. This technique then looks for the first subcomponent of the template that is connected to "Vout." In this case, the first p-type transistor in the template is selected. This technique then maps "Vout" to the first output node listed in the linked list in Figure 9, "node2." Now with "Vout" mapped to "node2," a match for the p-type transistor from the template is looked for. After this initial match is made, the remainder of the template is searched for the same way as in the first technique. Each output node in the list is checked for matches to the template. This technique does, however, have a severe draw back. Only components that meet the criteria of the output node can be extracted. At the first level of hierarchy this is not generally a problem. Figure 10 shows some of the common gates that meet this criteria. Since this technique is based on the outputs of the components, it is referred to as GENTEXO.

The two variations of each of the three techniques deal with the method of checking for a loss of connectivity. In the first variation, the original variation, the loss of

connectivity is not checked until after all subcomponents within the template are matched. The second variation detects the potential loss of connectivity with each match of the subcomponents.

In the second variation, the number of pins connected to each node is updated with the addition or deletion of any components in the data base. The number of pins connected

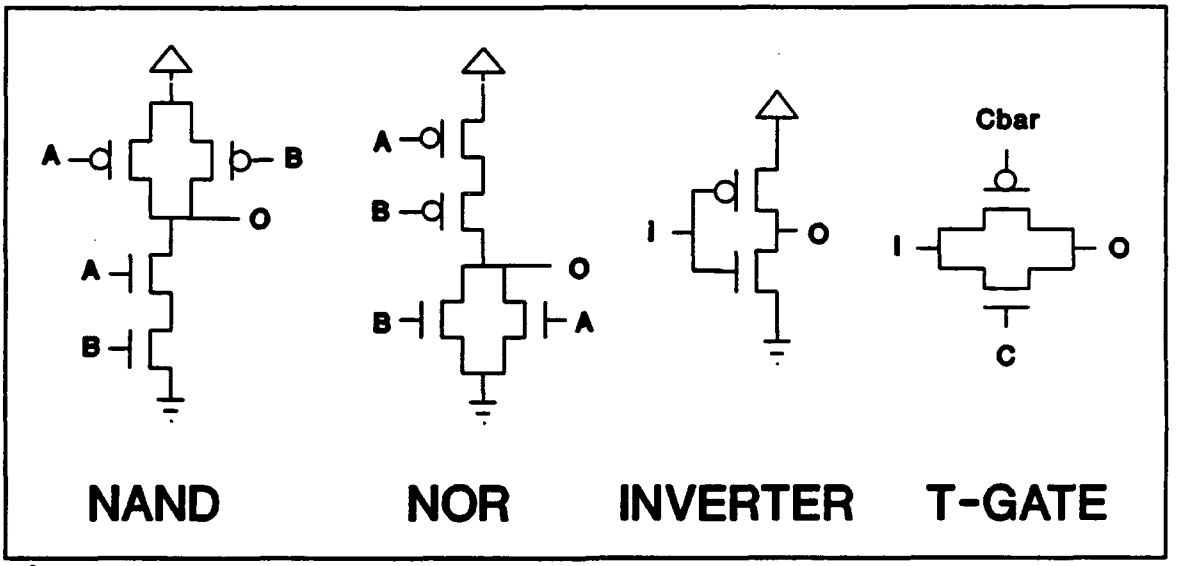


Figure 10. Some Common CMOS Gates.

to each of the internal nodes within the template are also calculated. Before the match of any new subcomponent is accepted, the node being mapped to the variable node name of any internal node within the template must have the same number of pins connected to it. For example, in Figure 5 there are some transistors that resemble a NAND gate but are not because of a connectivity problem at "node3". The first variation matches all four transistors of the template to this fake NAND gate. Not until after all the matches are

made is the connectivity problem detected and the match rejected. The second variation compares the number of pins connected to "node3" at the time it is encountered. From the template in Figure 6, it is known that any node that is correctly mapped to "VN1" must have exactly two pins connected to it. Node "node3" has four pins connected to it, so it is immediately rejected as a possible match. This check has two potential advantages. The first is that connectivity problems are detected earlier. The second is that other potential matches to incorrect subcomponents may be determined earlier in the extraction process. Since this variation checked the number of connections on the nodes at each step in the process, each of the three techniques that uses this variation has an "N" added to the end of its name.

The three techniques with the first variation are: 1) GENTEX, 2) GENTEXC, and 3) GENTEXO. The three techniques that used the second variation are: 1) GENTEXN, 2) GENTEXCN, and 3) GENTEXON.

3.5 Permutability

The permutability of pins has two possible solutions. These solutions are very different and both solutions can not be tested without extensive modifications to the source code. Time constraints did not allow this. For this reason, only one solution to this problem was tested.

The first solution is to write the extraction program such that the order of the pins of each component was fixed.

This means that the extraction program has to treat each permutation as a separate template. The number of templates in this case grows exponentially with the number of permutations and the number of subcomponents within the template. If M subcomponents, each with N permutable pins, are contained within a template then there are $(N!)^M$ possible templates [12]. For example, the template for the NAND gate consisted of four transistors. The drain and source of each transistor are permutable. Therefore, there are $(2!)^4$, or sixteen, required templates to match all combinations. If a template consists of five three-input NAND gates, the number of templates grows to 7776 templates. The number of templates can get out of control very quickly.

The second solution is to write the extraction program such that the order of pins of each component is flexible. This means that additional information is required in the template to identify the desired pin. Each pin within the template requires some type of identifying mark or characteristic. For example, the pins of an inverter can be marked as input and output, and the pins of a t-gate can be marked as bi-directional for two of the pins. Different identifying marks are required for the two control pins of the t-gate. This solution requires that the extraction routine first search the component for the correct pin before checking for proper connection with other components.

The first solution was chosen to be implemented in this

thesis for several reasons. First, it is simpler to implement. The high number of templates possible to find a single new component is generated internally by GENTEX. This relieves the user from this burdensome task. Second, it can not be determined which solution is faster without actually coding both solutions and testing them. Due to the hierarchical approach in extraction, the number of subcomponents within a template tends to stay relatively low. This reduces the number of combinations of the permutable pins. There is a definite overhead associated with the second solution. These factors require further study to determine the faster of the two approaches. A study, done on an expert system [12], showed that the first solution is not only faster than the second solution, but also uses less memory than the second solution. However, these results are not necessarily indicative of the results when written in the C programming language.

3.6 Interfacing With EDIF

The Electronic Design Interchange Format (EDIF), ANSII/EIA-548-88, is a national standard format to represent electronic designs. Most of the new software in VLSI supports EDIF or a subset thereof. EDIF 2 0 0 is the current version of EDIF. EDIF 2 0 0 is supported in the AFIT environment to some degree by SGE of Synopsys 2.2, CAPFAST 2.2, and most recently Cadence. EDIF is the only common format between these systems. This reason and the

fact that EDIF is the official national standard are why EDIF was chosen as the format for interfacing within this thesis.

An EDIF interface was required on both the input and output sides of GENTEX to allow for the greatest possible

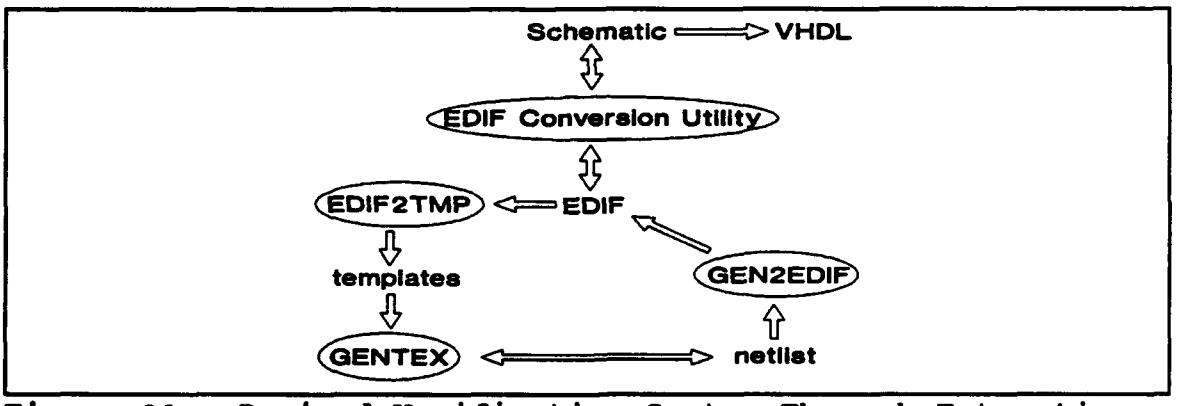


Figure 11. Desired Verification System Through Extraction.

amount of automation in the verification process. Figure 11 shows the new verification process, and Appendix A describes the programs used in the verification process in more detail. Schematics of what the designer expects to be in the transistor net list are converted from EDIF to the template format of GENTEX. The ".sim" file from MEXTRA or EXT2SIM is then extracted using the templates generated from the schematics. Hierarchical templates are generated with hierarchical schematics. The extracted file is then converted to a schematic in EDIF. This schematic is read into one of the schematic capture tools to verify its correctness. Since all components are placed in their relative positions with respect to their layout in Magic,

errors are more easily found.

When the contents of the transistor net list are unknown, the system shown in Figure 11 may be used to generate the VHDL code for it. The ".sim" file is extracted using GENTEX and a library of templates. The extracted file is then converted to a schematic in EDIF. SGE of Synopsys then generates the VHDL code for the schematic of the unknown circuit.

3.7 Data Collection

There were two types of data that was collected. The data collected from the EDIF interfacing was qualitative while the data collected from the extraction programs was quantitative.

The EDIF interfaces were tested on CAPFAST and SGE. The schematics in EDIF generated by GEN2EDIF (a translation program written as part of this thesis) were translated into the format of both schematic capture programs. The results of this process could only be the comparison of the quality and accuracy of the two translations. These results are in Chapter 7. To test the EDIF2TMP utility (another translation program written as part of this thesis), EDIF was generated by CAPFAST, Synopsys, and GEN2EDIF. The results again had to be based on the quality and accuracy of the translation. These results are also in Chapter 7.

The extraction algorithms were timed while extracting the exact same templates from the same net list. The times

were then compared in order to compare the algorithms. The time of each run was taken by the UNIX system call "time" with the format "time <command>" where "<command>" is any command that can be run from the command line. This program provides three times after the completion of the command. The three times are called real, user, and system. The real time was the total time from the start to the finish of the process. The user time is the total time the system worked on the process, better known as CPU time. Finally, the system time is the total time spent on system overhead for the process. For the purposes of this thesis, the user time (CPU time) was used. The real time was greatly affected by other processes running on the machine. There was theoretically no effect on user time if other processes were running on the same machine. This theory is examined in Chapter 5 along with the other results from the timing analysis.

3.8 Summary

This chapter explains and discusses the methods, approaches, and techniques being used in this thesis. The problems that were encountered in previous programs are discussed. Some possible solutions to these problems and the justification for the solutions implemented are discussed. The three techniques of circuit extraction and the two variations of those techniques, which account for the six extraction algorithms, are discussed. The names of

the six extraction algorithms (GENTEX, GENTEXC, GENTEXO, GENTEXN, GENTEXCN, GENTEXON) are explained. The reasons and alternatives to using fixed pin order for the components are discussed in relation to permutability. EDIF is the format of choice for interfacing the extraction routines with other AFIT systems. Finally, the methods of data collection are discussed. The EDIF interfaces had to be evaluated qualitatively, while the performance of the extraction algorithms could be evaluated quantitatively.

IV. GENTEX

4.1 Introduction

GENTEX (GENeric Template EXtractor) was started as a special study project. The purpose of GENTEX was to address some of the problems with previous extraction systems. GENTEX was written in the standard C programming language. This extractor was based on STOVE and therefore borrowed from it in several ways. The data structures and program flow used in both programs are similar. Some subroutines, such as the memory management routines, are also borrowed. Overall, STOVE contributed approximately fifteen percent of the original code of GENTEX, either directly or indirectly. With the modifications made to GENTEX in this thesis, the contribution by STOVE is less than five percent.

This chapter describes GENTEX in detail, including the some of the modifications that were made. Some of the problems that GENTEX attempts to solve are inherent to the language used. Some of the problems are inherent to the approach used.

4.2 Solutions to Previous Problems

GENTEX was written to solve as many of the problems previously encountered as possible. These problems can be grouped into two categories: templates and performance.

It was the goal of the templates to be as simple as

possible. This minimizes the problem of adding new components. The templates also had to allow permutability of pins to be specified. The problem of connectivity was addressed in the code and was therefore transparent to the user.

The performance problems addressed with GENTEX included speed and memory usage. Speed was the main concern. Six algorithms were tested to find the one with the best performance. After the fastest algorithm was determined, GENTEX was optimized for that algorithm to further increase speed. Memory usage was a secondary concern. The C programming language avoids many of the problems with memory usage that are inherent to most expert systems. Due to time constraints, memory usage was not optimized. Some memory management routines were used in the code, however, to avoid excessive memory usage.

4.3 GENTEX Program Flow

Figure 12 on the next page shows the high-level pseudocode for GENTEX. Some detail is shown in the read ".sim" file and template extraction routines, but for the most part, little detail is shown. The actual code for the six extraction programs is in Appendix B.

Figure 12 shows that the first thing that GENTEX does is parse the command line. This is where the command line flags and the input file are read and checked. If any errors are found at this point, a short message is displayed

```
main program() {
    parse command line and set flags;
    initialize global variables;
    if (not Tflag) then read template file;
    call read_sim_file();
    if ((not iflag) and (not bflag)) extract inverters;
    if ((not tflag) and (not bflag)) extract t-gates;
    if (not Tflag) call extract_templates();
    print results;  }   /* end main */

read_sim_file() {
    for each new component {
        get component type;
        for each pin of component {
            read node name;
            if node does not exist then create new node;
            add pin to linked lists;
            }
        }
    check for duplicates; }  /* end read_sim_file */

extract_templates() {
    for each template {
        subcomponent = first subcomponent of template;
        done = false;
        loop {
            find subcomponent;
            if (not found) then {
                subcomponent = previous subcomponent;
                if (subcomponent = NULL) then done = true;
                } /* end if */
            else {
                subcomponent = next subcomponent;
                if (subcomponent = NULL) then {
                  check connectivity;
                  if (connectivity = ok) then {
                    make new component;
                    delete subcomponents;
                    subcomponent = first subcomponent of template;
                    } /* end if */
                  } /* end if */
                }  /* end else */
            }  until done;  /* end loop */
        }  /* end for */
}  /* end extract_templates */
```

Figure 12. Pseudocode for GENTEX.

explaining the usage of GENTEX, and the program is exited.

After the command line has been read, the initialization routine is called. This routine initializes all the global variables and arrays, reads in the library, and reads the templates if a template extraction is not excluded by the "-T" option in the command line. If multiple templates need to be generated due to permutable pins of a component, that is also done at this point.

After initialization has been completed, the component net list is read. The input file is read line by line. If the first word in a line matches a valid component name in the library, it is processed; otherwise, it is ignored as a comment. The node structures and linked lists to the pins and components are generated as each component is read from the input file. After the file has been read, the circuits are checked for duplicate components. If any duplicates are found, one is deleted and the width of the remaining component is increased by the width of the deleted component.

GENTEX then searches for and extracts any inverters or t-gates in the data if these extractions have not been excluded by the "-i," "-t,' or "-b" options in the command line. The inverters are done first and then the t-gates. No templates are required for these two new components; they are hard-coded. Hard-coding the extraction of inverters and t-gates, the two most common components used in CMOS,

significantly decreased the time required for extraction.

The next part of the program is the template extraction. This is the most important part of the program. The pseudocode in Figure 12 shows in a little more detail the operation of this part of the program. The most important lines in Figure 12 are "find subcomponent;" and "check connectivity;" in the template extraction routine. These two lines represent the focus of this thesis. The difference in the six extraction algorithms is in the way these two lines are accomplished. The templates are searched for in the order given in the template file. Any matches are immediately extracted and the length, width, x-position, and y-position are updated for the new component. The length and position of the extracted component is equal to that of the first subcomponent found within the template. The width of the extracted component is equal to the sum of all the widths of the subcomponents used to make the match to the template.

After the extraction is complete, GENTEX sends the results to the output file and prints the final statistics on the screen. The output file is by default the input filename with a ".out" extension replacing the ".sim" extension. The output file name can be changed, however, with the use of the "-o" option in the command line.

4.4 The Template

The template is an important part of GENTEX. Templates

define new components in terms of a pattern of other components. For example, a NAND gate is made up of two "p"s and two "e"s. However, only the specific combination and interconnection of the four transistors shown in Figure 3 constitute a NAND gate. The template, like the ones shown in Figure 13, are an easy way to give this information to GENTEX. The templates are located in a file called "gentex.tmp."

Format. The format of the template is quite simple. Only the lines between "begin template;" and "end template;" are considered to be part of the template. Everything else is a comment. This means that the template may be preceded

```
/*The following is the template for a NAND gate*/
begin template;
    nand VA VL VO;
    p VA LVdd:1 VO:1;
    p VL LVdd:1 VO:1;
    e VL VO:1 VX:1;
    e VA VX:1 LGND:1;
end template;

/*The following is the template for an AND gate*/
begin template;
    and Vinput1 Vinput2 Voutput;
    nand Vinput1:1 Vinput2:1 Vtemp_node;
    inverter Vtemp_node Voutput;
end template;
```

Figure 13. Templates for a NAND Gate and an AND Gate in GENTEX.

or followed by comments. However, after "begin template;", the next line names the new component. So, if a comment is placed in between the "begin template;" and the intended new

The remainder of the template defines the component and the template ends with "end template;". It should be noted that the templates are case sensitive, including the begin and end statements.

Comments may be placed within the template as long as they follow a few rules. Comments may be placed outside the template unconditionally. Within the template, comments may be placed on the same line as a circuit description as long as they are after the semicolon after the last node name. Comments may start a new line as long as the first word in the comment does not appear in the library and the comment is not immediately preceded by "begin template;." It is good practice to be consistent with the comments to avoid confusion. Comments should be set off by "/*comment*/" or preceded by "--" or some other method.

Node Names. There are two types of node names. The first is the literal node name. Literal node names are used when a specific node is desired. The first letter in all literal node names must be "L". For example, if a pin needs to be connected to ground, the node name is "LGND".

The second type of node name is the variable node name. Variable node names are used when no specific node is required. All variable node names must begin with "V." For example, if a node does not need to be connected to any specific node, its node name could be "Vinput".

The node names are used to define a specific pattern to locate in the data structures. Pins connected to the same node must have the same node name. In the NAND gate example, there are six unique nodes. The resulting NAND gate has three pins. Figure 13 shows the template for the NAND gate.

There were several things to note about the NAND template example. First, the last node name ends with a semicolon. Also notice that any leading spaces before the component name are ignored. If a semicolon is not found, then GENTEX continues with the next line as if it were part of the previous line. This means that circuits with a lot of pins do not need to be completely on one line. Finally, it is important to remember that the order of the given node names is important. GENTEX reads the template as order specific. This means that the first node name may only match a pin that is the first pin in a component. The order of the pins of "e"s, "p"s, "inverter"s, and "t-gate"s is already defined. The order of the pins of a new component is defined by the order of the pins in the template. Figure 14 shows the defined pin orders of the hard-coded component types.

Maps. This introduces another potential problem. Since the node names are order specific and the drain and source of a transistor are interchangeable, or permutable, the template would not recognize a NAND gate whose first "p"

e	gate drain source
p	gate drain source
inverter	input output
t-gate	input (gate of 'p') (gate of 'e') output

Figure 14. Pin Order for Existing Components in GENTEX.

happened to have had "Vdd" connected to the source and "out" connected to the drain. One solution is to write another template. This, however, can quickly get out of hand. Sixteen different templates must be written to accommodate every possible combination of the sources and drains of the four transistors in a NAND gate. The problem grows even more rapidly with three or more input devices such as a three-input NAND gate.

The solution used in this thesis is to mark the permutable pins and have GENTEX generate templates for every possible combination of the pins. This can still get out of hand, but it is not seen by the user.

The chosen approach to the problem requires that the permutable pins be marked in some way within the template. This is where the map field comes into play. Each node name in the template may optionally end in a colon followed by a number. Pins of a component that have the same number in the map field are permutable. Pins that have unique numbers, the number zero, or no number are considered fixed. Pins are only permutable within the same component so the

same map field number may be used in different subcomponents. For example, the third line in Figure 13 is "p VA LVdd:1 VO:1;" and shows that the two pins "Vdd" and "VO" are permutable. The drain and source are marked as permutable. In fact, the drain and source of all four transistors as well as the inputs of the NAND gate (in the definition of the AND gate) in Figure 13, are marked as permutable.

Finally, it is acceptable to define a template and then use that new component as a subcomponent in a later template to define another component. For example, Figure 13 shows the template for an AND gate. In CMOS, an AND gate is a NAND gate followed by an inverter. Figure 13 shows how the NAND gate is first defined and then used later in the definition of the AND gate. The AND gate may then be used in a following definition of another component. This process can continue until the desired level of abstraction is reached.

4.5 The Library

The library has information on component names, their associated number of pins, the current number of each component type in the data base, and a unique number denoting that component type.

The unique number is based on when the component was added to the library. The first component added is numbered one, the second two, etc. This is done to decrease search

times in the extraction routines. To insure the component is of the proper type, the integers are compared rather than doing a string compare to check names.

There are three ways that a circuit is entered into the library. The first way is to be hard-coded. GENTEX has four components hard-coded in the library: "p", "e", "inverter", and "t-gate". These four cannot be changed without changing the source code and recompiling. Note that "n" was not a listed type. Anytime the component type was found to be "n," it was changed to "e" within GENTEX so there would not be any problems with confusing the two since they are equivalent for the purposes of this thesis.

The second way for a component to enter the library is to be included in the library file. The library file is "gentex.lib". If this file exists, it is read line-by-line. Each line represents a new library entry. The component name is expected to be the first item on the line, immediately followed by the associated number of pins (e.g. "NAND 3"). Any desired comments must be on the same line as the new library entry following the number of pins. Everything on the line following the number of pins is ignored, so no specific format is required for the comments; however, consistency is recommended.

The final way for a component to be entered into the library is to be defined by a template. If the new component being defined by a template is not currently in

the library, it is added. For example, if a template defining a NAND gate is given, it is not necessary to put "NAND 3" into the library file. When the template is read and it is realized that the new component is not yet in the library, it is added to the library and the number of pins equals the number of node names given.

4.6 Summary

This chapter describes GENTEX in detail. The methods used by GENTEX to solve some of the problems with previous extraction programs are discussed and explained in detail. GENTEX is written in the C programming language. C does not have the inherent problems with memory usage and performance associated with artificial intelligence programming environments. The map field in the templates is used to denote permutability of pins. This problem is generally ignored by other extraction programs.

V. Results of Extraction Programs

5.1 Introduction

This chapter presents and discusses the results of the extraction programs. These results include the problems encountered, an analysis on the precision of the recorded times, the relative performance of the extraction algorithms, the possible effects on performance of changing certain aspects of the templates, and finally a comparison between GENTEX and other extraction routines.

5.2 Problems/Limitations

The focus of this section is on the problems or limitations encountered that in some way limit the ability of the extraction programs to verify circuits. The first problem was with cell permutability. GENTEX allows individual pins to be permutable, but in some cases this was not enough. Eight memory cells are grouped together to form a word. Electrically, these cells do not have a specific order until other connections are made to them. However, the order in which GENTEX extracts the cells may not be the order in which they were physically connected. So when trying to line up rows or columns of groups of memory cells, the cells are not in the same order. The cells are permutable while the individual pins were not permutable. If one pin is exchanged, then the entire cell has to be

exchanged. GENTEX does not have a mechanism to allow this. The problem had to be by-passed by using the literal node names from the actual component netlist in the templates for memory.

The next problem was with an arbitrary limit GENTEX put on the number of templates allowed and the number of pins on a component. GENTEX requires a declaration of the maximum number of pins on a component and the maximum number of templates (including permutations) before the program can be compiled. The arbitrary limits were too low and were raised to 2,000 pins and 10,000 templates. The code could be re-written in such a way to eliminate the need for these arbitrary limits, but time constraints prevented it.

The last problem encountered with the extraction program was due to the technique used in GENTEXO and GENTEXON. It was known up front that this method would be limited to complementary gates. However, the number of components and gates that do not meet the criteria for GENTEXO and GENTEXON was unexpected, especially in a CMOS circuit. A lot of special preparation was required in order to create performance tests that met the requirements of GENTEXO and GENTEXON that were fair and unbiased towards any of the extraction algorithms.

5.3 Timing Precision

The entire analysis of the six extraction algorithms was based on the times produced by the "time" system call.

Therefore, it was necessary to test this program for its precision. There is a difference between precision and accuracy. The program is precise if it yields nearly the same times with each run of the same command. The program is accurate if the times produced by the program are close to the true value. The accuracy of the program could not be tested because the actual running times of the commands were not available.

It was important to establish that the times were precise under all conditions. If the times were not precise, it would be necessary to run each extraction several times and then take the average. This would present a problem for some of the extractions which took several hours. It was also important to determine whether the workload on the computer affected the times. If the workload did significantly affect the times, then the same workload on the computer would be required for every extraction for the duration of the extraction. This would have made it nearly impossible to take accurate data.

Three test cases are shown in Table I to demonstrate the precision of the timing program. Each test case was run twice while the computer was under a heavy workload and twice while the computer was under a minimum workload. Many other similar tests were also performed and yielded similar results. The data in Table I shows that the greatest difference in times was less than one percent. The data

Table I. Data for Timing Precision Analysis.

	Case 1 (mm:ss)	Case 2 (mm:ss)	Case 3 (h:mm:ss)
run 1 (min workload):	1:56.0	21:56.5	1:16:09.9
run 2 (min workload):	1:55.8	21:51.0	1:16:09.1
run 3 (max workload):	1:55.5	21:51.3	1:16:14.4
run 4 (max workload):	1:55.7	21:51.5	1:16:09.9
difference between max and min times:	0.43%	0.42%	0.12%

also shows that the times were independent of the workload on the computer. Based on these results, the precision of "time" was satisfactory for the purposes in this thesis.

5.4 Examples

The data and comparisons of this thesis would not be valid if the results of the extractions were not correct. Three examples were used to verify that the extraction programs were working correctly. Important information regarding these three tests is located in Table II.

The first example is an eight bit adder/subtractor. A schematic was provided by the designer of the adder/subtractor that had been verified to be correct as part of a class project. This circuit was extracted to eight full adder/subtractors based on the schematics provided. The extracted circuit was given to the designer who verified that the eight remaining components were connected properly.

Table II. Important Data from the Three Test Cases of the Extraction Programs.

	Case 1	Case 2	Case 3
components at start:	384	19,480	141,871
components at end:	8	859	414
inverters found:	72	2926	34148
extraction time:	0:00:00.2	0:01:17	0:30:32
t-gates found:	48	2968	243
extraction time:	0:00:00.1	0:01:00	note 1
duplicates found:	0	0	note 2
extraction time:	0:00:00.2	0:02:01	note 3
# of templates:	4	5	30
template permutations:	468	35	217
extraction time:	0:00:04.5	0:04:56	0:53:22
memory used (Meg):	2	3.5	29
total time:	0:00:05.0	0:09:14	5:57:30

All times are in hh:mm:ss format.

note 1: The t-gates were not extracted immediately after the inverters as is usually the case. Extraction time was 0.3 seconds.

note 2: The initial number of duplicates found was 1506, but due to the permutation of the source and drain, templates had to be written to find the remaining 148 duplicate transistors.

note 3: The time of the original search was 1:54:08. The time of the template search of the duplicate transistors was 2:40:28.

The second example was the mantissa circuitry for a IEEE double-precision floating-point multiplier developed by L. Kesting. ESIM and HSPICE were used alongside with GENTEX and GEN2EDIF to validate and verify the circuit. Not only were the results of the extractions compared with the

results of the simulators, Kesting also spent considerable time manually verifying the results of the extractions by visually comparing the results with the transistor layout. Several common errors were found and avoided in the mantissa transistor layout.

Finally, the third and last example used to test the extraction routines was a portion of a specialized memory chip. The portion used consisted mostly of eight transistor memory cells. In fact, the 2Kx8 bit memory accounted for 131,072 of the 141,871 transistors in the circuit. The designer of this circuit, S. Sangregory, provided schematics of the circuit and its subcomponents. Sangregory used HSPICE, ESIM, and CSTAT (a connectivity checker for CMOS chips developed at AFIT) to validate the circuit. The extraction results were returned to and visually verified to be correct by Sangregory.

5.5 Algorithm Comparison

The main focus of this thesis was to examine the performance differences between the six extraction algorithms. This section summarizes the data collected to perform this comparison. The times given are the times actually spent extracting the components. A separate run on each circuit was made to determine the time required to read and write the circuit (the overhead). This time was then subtracted from the times of each extraction run to yield the time spent extracting.

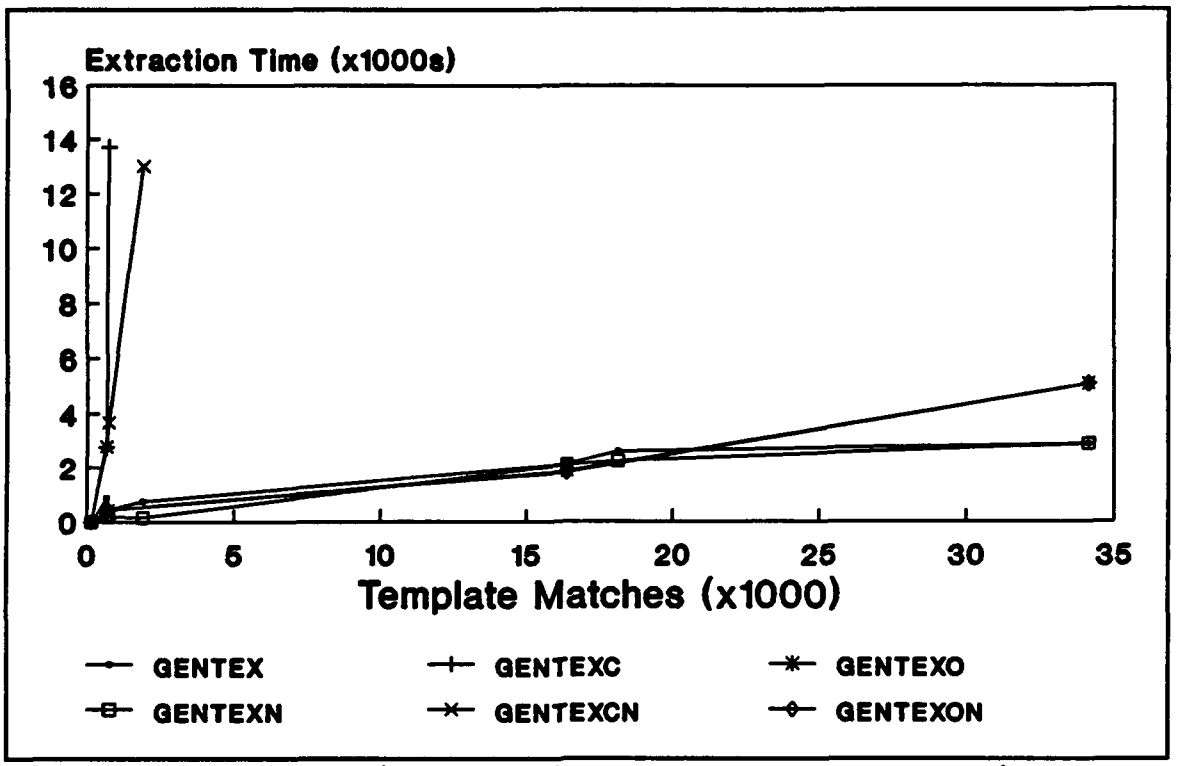


Figure 15. Extraction Algorithm Performance Comparison.

The extraction times were dependent on a number of variables. The size of the circuit being extracted was an important variable. The number of templates and number of matches to those templates was also important. Other variables such as: type of templates, order of templates, size of templates, and the order within the templates were all important factors in extraction times. Unfortunately it is impossible to keep all variables constant for every performance measurement. Figure 15 shows a performance comparison based on only one of these variables. However, all variables were taken into account when the following conclusions were made.

The graph in Figure 15 shows a surprising result.

GENTEXO and GENTEXON did not consistently out-perform the other four algorithms. In fact, based on the data in Figure 15 and some data not shown, these two algorithms got relatively slower for larger extractions. This was completely unexpected since these algorithms had the advantage of knowing the possible output nodes before the search began. The best possible explanation for this result is that the overhead required for such a search is much larger than suspected for these type of circuits.

Figure 15 shows that the approach used in GENTEX consistently out-performed the approach used in GENTEXC and that GENTEXO generally out-performed both of the other two approaches. However, the effect of the "N" variation on the three approaches varied somewhat. In general, the "N" variation of GENTEX (GENTEXN) was significantly faster then GENTEX. The speed-up obtained by the "N" variation on GENTEXO was generally insignificant. Finally, the speed-up of the "N" variation on GENTEXC varied greatly from circuit to circuit. Unfortunately, the graph shows GENTEXN and GENTEX as well as GENTEXON and GENTEXO converging at a high number of template matches. This is due to using an inverter template to generate a high number of matches. Inverters have no internal nodes and therefore the "N" variation and the original variation of the programs converge on these extractions. In general, the "N" variation is faster and never significantly slower.

GENTEXC and GENTEXCN performed very poorly except for on very small circuits. The times required to extract components went up exponentially with the number of components in the circuit. These algorithms perform linear searches through every component of a particular type where GENTEX and GENTEXN only check the components on a particular node. Tests show that GENTEXC and GENTEXCN generally outperform the other algorithms on very small circuits (approximately 500 components or less). However, even though the speed-up for smaller circuits may be a factor of two or three, the times involved are generally less than five seconds. For larger circuits, GENTEXC and GENTEXCN quickly drop off in performance.

The extraction times of GENTEX, GENTEXN, GENTEXO, and GENTEXON increase linearly with the number of components in the circuit. The increase appears to be faster in GENTEXO and GENTEXON. The type of circuit used in the last two test cases may have had an important part of this. GENTEXO appears to be faster than GENTEX for circuits less than about 100,000 components depending on the templates used. GENTEXON appears to lose its speed advantage on GENTEXN on circuits of approximately 5,000 components depending on the templates used.

5.6 Multiple Extractions

It had been shown that the number of rules (templates) in an expert system increased the processing time

exponentially rather than linearly [13]. It was shown that by breaking the rules up into several files and extracting the files separately that performance could be increased for the expert system [13]. The sum of the times for the partial extractions is less than the time required for the total extraction.

Table III. Single Extraction Versus Multiple Extractions.

	run 1 (mm:ss)	run 2 (mm:ss)	run 3 (mm:ss)	total (mm:ss)
GENTEX				
1 piece	43:18	-----	-----	43:18
2 pieces	35:24	8:24	-----	43:48
3 pieces	35:24	7:42	0:45	43:51
GENTEXN				
1 piece	37:09	-----	-----	37:09
2 pieces	35:01	0:59	-----	37:00
3 pieces	35:01	0:40	0:23	36:04

It was necessary to determine if this held true for the GENTEX algorithms. Table III shows an example of a large number templates and the extraction times for GENTEX and GENTEXN for a various number of extractions. This table shows that the total times are approximately equal despite the number of runs to do the extractions.

5.7 Varying the Order of Templates

It was suspected that varying the order in which the templates were extracted could affect the times required for the extraction. The idea was that if the template with the

greatest number of matches was searched for first, there would be less components remaining for the search of the next template. In this manner, the number of components would decrease more rapidly than if templates with less matches were searched for first.

Table IV. Effect of Varying Template Order on Extraction Times.

	Case 1	Case 2
components at start:	19,480	140,217
components at end:	859	18,343
number of templates:	5	11
template permutations:	35	155
Order of most to least extraction times:		
GENTEX:	0:12:26	1:16:38
GENTEXN:	0:02:26	0:34:46
Order of least to most extraction times:		
GENTEX:	0:17:07	note 1
GENTEXN:	0:07:18	16:13:58

All times are in hh:mm:ss format.

note 1: This extraction was stopped after 36 hours and 29 permutations of templates.

Table IV shows the extraction times for GENTEX and GENTEXN in two cases where templates were extracted in an order based on the number of matches. The data shows that both algorithms show significantly slower times when the order of the templates goes from least to most number of

matches. The data also shows that the greater the difference in the number of matches, the greater the difference in extraction times.

5.8 Varying the Order within the Templates

It was also suspected that the order of the subcomponents within the template could affect the extraction times. All six of the extraction algorithms find matches to the templates component by component based on the subcomponents of the template. Therefore the order of the subcomponents within the template would certainly affect the extraction times, but the magnitude of this effect required further study.

An XNOR gate is a good simple example of the effect of the order of subcomponents within a template on the

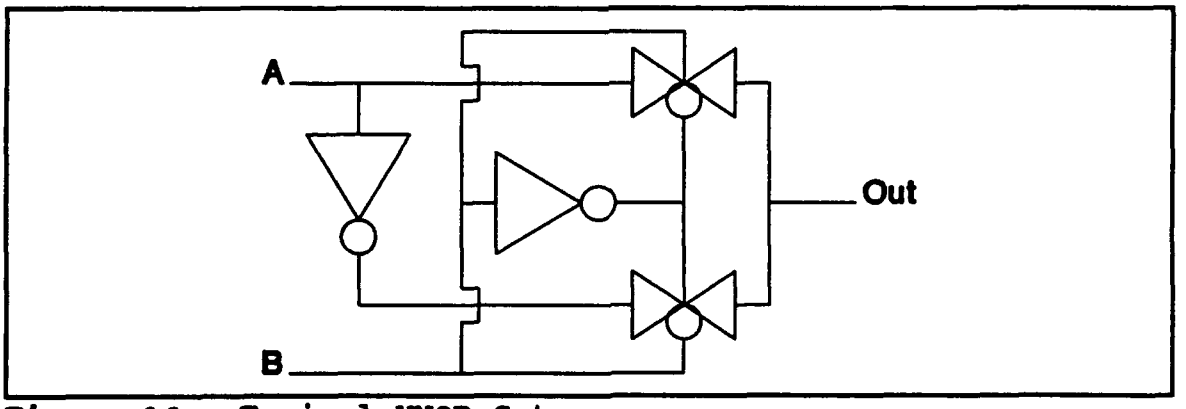


Figure 16. Typical XNOR Gate.

extraction times. Figure 16 shows the schematic of a typical XNOR gate. Notice that the inverters have no common nodes. This means that if the inverters are searched for

first, every pair of inverters will meet the requirements within the template. It will not be until the first t-gate is searched for that potential matches will be rejected. On the other hand, the t-gates have three nodes in common. Potential matches can be rejected with the search for the second t-gate. Table V shows some extraction times of XNOR gates with the inverters first versus searching for the t-

Table V. Comparison of the Order of Subcomponents Within the Template.

	Case 1	Case 2
components at start:	264	13586
components at end:	192	13586
Extraction times with t-gates of XNOR gate searched first:		
GENTEX:	0:00:00.1	0:00:03.2
GENTEXN:	< 0:00:00.1	0:00:00.7
Extraction time with inverters of XNOR gate searched first:		
GENTEX:	0:00:03.1	1:26:02
GENTEXN:	0:00:00.8	0:07:22

All times are in hh:mm:ss format.

gates first.

There are two results of the data in Table V. The first is that the order of the subcomponents within the template affects the extraction times a great deal. The second result is that GENTEXN is affected much less than

GENTEX. The "N" variation in GENTEXN, GENTEXON, and GENTEXCN help reject potential matches earlier in the matching process. If a node does not have the correct number of pins connected to it, it is rejected immediately. The original variation of the three extraction approaches waits until all the subcomponents of the template have been found before checking for the loss of connectivity. This allows potential matches to a template to get further into the matching process before being rejected. This is a big advantage of the "N" variation.

5.9 Lessons Learned

There were several lessons learned from the analysis of the performance data. Some of the lessons learned were incorporated in the final product discussed in Chapter 6. However, perhaps the most important lessons learned for a user of an extraction program dealt with the templates. It was shown that the templates may affect performance more than the extraction algorithm. Therefore, it is very important to discuss the art of writing templates.

The concept behind writing efficient templates is simple. Make incorrect matches to the template fail as quickly as possible. The practice of this concept, however, is not straightforward. There were three basic rules discovered in this thesis that aid in writing efficient templates. The following three rules are discussed in the order of their importance under most circumstances.

Literal Names First. The first rule to writing efficient templates is to put subcomponents of the template that have a literal node name first. For example, a template is written for a NAND gate. Three of the four transistors have a connection to either "Vdd" or "GND". One of these three transistors should be listed first. The literal node name cuts down the possible matches to that subcomponent of the template to components on the specific node only. If a subcomponent with all variable node names is listed first, every component in the circuit of the same type will match that subcomponent of the template.

List the Subcomponents in Order of Connection. The second rule to writing efficient templates is to make sure that each subcomponent listed in the template has a node name that has a common node to a subcomponent listed previous to it. For example, a template is written for the XNOR gate in Figure 16. If an inverter was listed as the first subcomponent of the template, the next component listed should not be the other inverter. The inverters have no common connections. This allows every pair of inverters to meet the criteria specified by the template. If a t-gate was listed second, there would be at least one common node. This would force the extraction routine to search the specific node that is mapped to the variable node name for the t-gate. In the case where the two inverters are first, no node names are mapped to specific nodes, and therefore

the search is not limited to a specific node and takes much longer to complete.

Least Common Subcomponent First. The last rule to observe when trying to write efficient templates is to try to make the first subcomponent in the template the least common of the components of the circuit. For example, a template for an AND gate is written to be a NAND gate followed by an inverter. The number of inverters in the circuit is probably much greater than the number of NAND gates, and therefore the NAND gate should be listed first in the template.

5.10 Performance Comparison

The final question that needed to answered in this thesis was "How does the performance of GENTEX compare with other extraction programs?" A good performance comparison was almost impossible due to the differences in the systems used and the data available. The following is a comparison of the GENTEXN algorithm (generically called GENTEX) and several other extraction programs including the advantages and disadvantages of each. These comparisons were done before the optimizations to GENTEXN discussed in Chapter 6 were made. More about each of the following programs can be found in Chapters 2 and 3.

GES. The first extraction program that needed to be examined was GES. The times reported in Captain Dukes' dissertation for a 250,000 transistor design running on a

MicroVAX 3900 are impressive. Applying a technique which he referred to as indexing, the circuit is extracted in approximately one hour. Inverters, t-gates, and clocked-inverters are the only components extracted. He also reported a time of approximately 4.5 hours to find and remove duplicate transistors. The first thing to note is that the Sun SPARC II on which GENTEX was run is faster than the MicroVAX 3900. The second thing to note is that the inverter and t-gate extractions in GENTEX as well as the duplicate search are hard-coded in C (and therefore should be faster) and, therefore, are not a reflection on the speed of the extraction algorithm.

The reported performance of GES was broken down into stages. GES found and removed duplicate transistors in a 250,000 transistor circuit in nearly the same time that it took GENTEX to remove duplicate transistors in a 140,000 transistor circuit. GENTEX even had the advantage of the faster computer and the algorithm was hard-coded in C where GES is coded in Prolog, an expert system. The inverters were extracted by GES in approximately 13 minutes. GENTEX took over 30 minutes to extract the inverters of the smaller circuit. Again, GENTEX had the advantage of the faster computer and the extraction routine was hard-coded C. There are two possible explanations for GES outperforming GENTEX under these conditions. The first is that these particular routines in GENTEX are coded inefficiently. The second is

that the circuits are very different. The possibility of Prolog significantly outperforming a well-written compiled C program is unlikely.

In comparison, the reported performance of GES was better than that of GENTEX. However, GES did not consider the permutability of pins except at the transistor level [9]. GES has many options such as VHDL output, critical path analysis, and timing information that are not available in GENTEX. The templates in GENTEX are much easier to write, and the EDIF conversion utilities provide interfacing to other CAD systems.

VERA. The next extraction program that was examined was VERA. VERA was also coded in Prolog. VERA was used to hierarchically extract the circuit of a compact disc error corrector [11]. This circuit consisted of 140,049 transistors with almost 128,000 of the transistors being part of random access memory. This circuit and the level of extraction was very similar to the 140,000 transistor circuit used to test GENTEX.

The extraction time for VERA on this circuit was nearly two hours on an APOLLO DN4500 [11]. This computer is slightly slower than the Sun SPARC II used by GENTEX. No time was available for the removal of duplicate transistors. Including the extraction of the inverters and t-gates, GENTEX extracted the circuit in less than 1.5 hours. Considering the difference in the two computers, these times

are comparable. It would be hard to say which program was actually faster.

Although the performance of VERA and GENTEX are nearly the same, GENTEX has one significant advantage. The templates used by GENTEX took less than three hours to write by hand. The data required by VERA for the extraction took over twelve manweeks to produce [11].

WOMBAT. The final extraction program examined for a performance comparison with GENTEX was WOMBAT. WOMBAT was developed using INTERLISP-D, an expert system, on a XEROX 1108 [12]. The performance difference between the Sun SPARC II and the XEROX 1108 is significant but not known precisely. Also the level of extraction and the type of circuit extracted were not available. This makes any comparison based on the information given speculative. WOMBAT, like GENTEX, had more than one extraction routine. The fastest times were used for this comparison.

WOMBAT extracted a 2,090 transistor circuit in 76.7 seconds. This is slightly greater than the amount of time required by GENTEX to extract a 20,000 transistor circuit. Based on the information given, it appears that GENTEX was faster even when accounting for the difference in computers. This is not surprising since WOMBAT was not created for performance, but rather to compare permutability issues.

5.11 Summary

This chapter discussed the results associated with the

extraction routines. Some problems and limitations with the extraction programs were presented. The precision of the reported times was examined and found to be satisfactory. Three test cases were used to demonstrated that the GENTEX algorithms were producing correct results. A comparison was made between the six extraction algorithms. GENTEXN was found to generally be the fastest, and GENTEXC was found to generally be the slowest. Unlike most expert systems, the GENTEX algorithms were found not to have any performance decrease by extracting all the templates in one run. Other systems break the template files into several pieces and run the extraction program several times in order to increase performance. This was found to be unnecessary for the GENTEX programs. If the templates were ordered such that the first template was the one that would have the highest number of matches, the performance of the extraction was greatly improved. The order of the subcomponents within the template was also found to be an important factor in performance. The subcomponents should be ordered such that an incorrect match would fail as quickly as possible. Finally, the performance of GENTEXN compared with other extraction programs found that GENTEXN was generally as fast or faster, and GENTEXN generally had other advantages. GES was the only exception. GES with indexing had only a slightly longer extraction time for a larger circuit on a slower computer.

VI. Modifications and Improvements

6.1 Introduction

This chapter discusses the modifications to the extraction programs that were not included with the results in Chapter 5. These modifications were not included in all the extraction algorithms due mostly to time constraints. In this chapter, a more memory efficient version of GENTEX is discussed. Also, and most importantly, the final product of this thesis, a modified GENTEXN, is discussed. Finally, a new algorithm for optimally reducing the complexity of templates is presented and discussed.

6.2 Initial Memory Tests

Early in this thesis, a more memory-efficient version of GENTEX was created. Unfortunately, at that time the GENTEX algorithm was expected to have the best overall performance so the modifications were made to that program. This memory-efficient version of GENTEX was called GENTEXM.

GENTEXM is a significant modification to GENTEX but the relative effect on memory usage is, for the most part, insignificant. There is only one modification in GENTEXM. The memory used by components that are deleted due to extraction is reused instead of being wasted. The greater the number of template matches, the more memory that is saved.

Significant amounts of memory are saved only when there are a large number of template matches. There are thirty-two bytes of memory saved for every new component plus an additional twenty bytes for every pin of that new component. This means that the amount of memory saved would be the same for all six extraction algorithms for a given circuit. In the case of the memory circuit (Case 3, Table II), 3 megabytes of memory were saved. This made the total memory usage 26 megabytes. This difference was fairly significant. However, in the other two test cases in Table II, the difference in memory usage was hardly noticeable.

Perhaps the most important result of GENTEXM was its effect on performance. No noticeable decrease in performance resulted. In fact, in some cases performance was actually improved due to the lower number of requests for additional memory to the operating system. In general, the effect on performance was less than one percent.

6.3 Performance Improvements

After it was determined that the GENTEXN algorithm had the best performance, the program was optimized for that particular algorithm. There were several things included in the program that were for either instrumentation purposes or were generic constructs to accommodate all six algorithms. These were for the most part removed from this modified version of GENTEXN.

It should first be noted that there were many subtle

coding changes that significantly improved performance. For example, one statement of the form, "if (A and B)" was changed to "if (B and A)." This change produced nearly a thirty percent increase in performance in that routine. Many of these types of changes were made.

There were several statements that printed intermediate results for the purpose of data collecting. Most of these extra statements were removed.

There was also a data field that could be removed from the program. This data field was hardly used and unneeded. It was included at the early stages of development of the program to accommodate a method of extraction that was later rejected.

The biggest modification in the program was the removal of a linked list. All the circuits of the same type were part of a unidirectional linked list as in Figure 8. It was suspected that the overhead required to maintain this linked list was too costly. The GENTEXC and GENTEXCN algorithms required this linked list, but the other four algorithms could operate without it. The removal of this linked list primarily affected five main routines within the program: 1) duplicate search, 2) inverter extraction, 3) t-gate extraction, 4) delete component, and 5) output.

The duplicate search routine had to be totally re-coded. Since the routine was being re-coded anyway, some other changes were made. Never during the course of this

Table VI. Before and After Data of GENTEXN Modification.

Extracted Component	Case 1 (seconds)		Case 2 (m:ss)		Case 3 (h:mm:ss)	
	Before	After	Before	After	Before	After
Duplicates	0.2	0.2	2:01	0:46	4:34:36	0:43:11
Inverters	0.2	< 0.1	1:17	0:08	0:30:32	0:02:21
T-gates	0.1	< 0.1	1:00	0:02	< 0:00:01	< 0:00:01
Templates	4.5	4.4	4:56	4:51	0:53:22	0:15:57
Total time	5.0	4.7	9:14	5:47	5:57:30	1:01:29

thesis was it necessary to search for duplicate components other than transistors. Therefore the duplicate search in the new routine was limited to transistors. Since the search was being limited to transistors, the code was able to be specialized specifically to transistors. Also, the permutability of the source and drain were incorporated into the new routine. The data in Table VI shows the significant increase in performance.

The inverter extraction routine was the least affected by the removal the linked list of component types. However, the changes required due to this removal plus some subtle changes similar to the ones described above, required that about half of the routine be re-coded.

The t-gate extraction routine required a major modification. The changes to the t-gate extraction routine was so extensive that it was basically rewritten.

The delete component routine was the main reason for

removing the linked list. It was suspected that removing a component from this list was too time consuming. The modification to this routine was almost as simple as removing the code that updated that particular linked list. Most of the performance increase shown in Table VI (except for the new specialized duplicate transistor search) is probably due to the decrease in time within this routine.

Finally, the output routine had to be totally re-coded. It is suspected that this change may have actually hurt performance slightly. However, the increase in time to output the results is on the order of seconds while the time saved in the other routines is on the order of minutes to hours.

Table VI shows the dramatic increase in performance for the optimized version of GENTEXN over the original GENTEXN. These new extraction times also make GENTEXN faster than GES. Chapter 5 stated that based strictly on reported extraction times, GES outperformed GENTEXN. After these modifications to GENTEXN, it can now be said that GENTEXN appears to perform as well if not better than GES in extraction times. However, there are other advantages and disadvantages of GES which are discussed in Chapters 2, 3, and 5.

Not shown in Table VI is the difference in memory usage. The modified GENTEXN used 27 megabytes of memory. Two megabytes less than previously used. If the memory

techniques discussed earlier in this chapter were applied, the total memory usage could be reduced to approximately 24 megabytes. This is a significant decrease in memory usage. The memory usage could be drastically reduced even more. The techniques required to do so are discussed in Chapter 8.

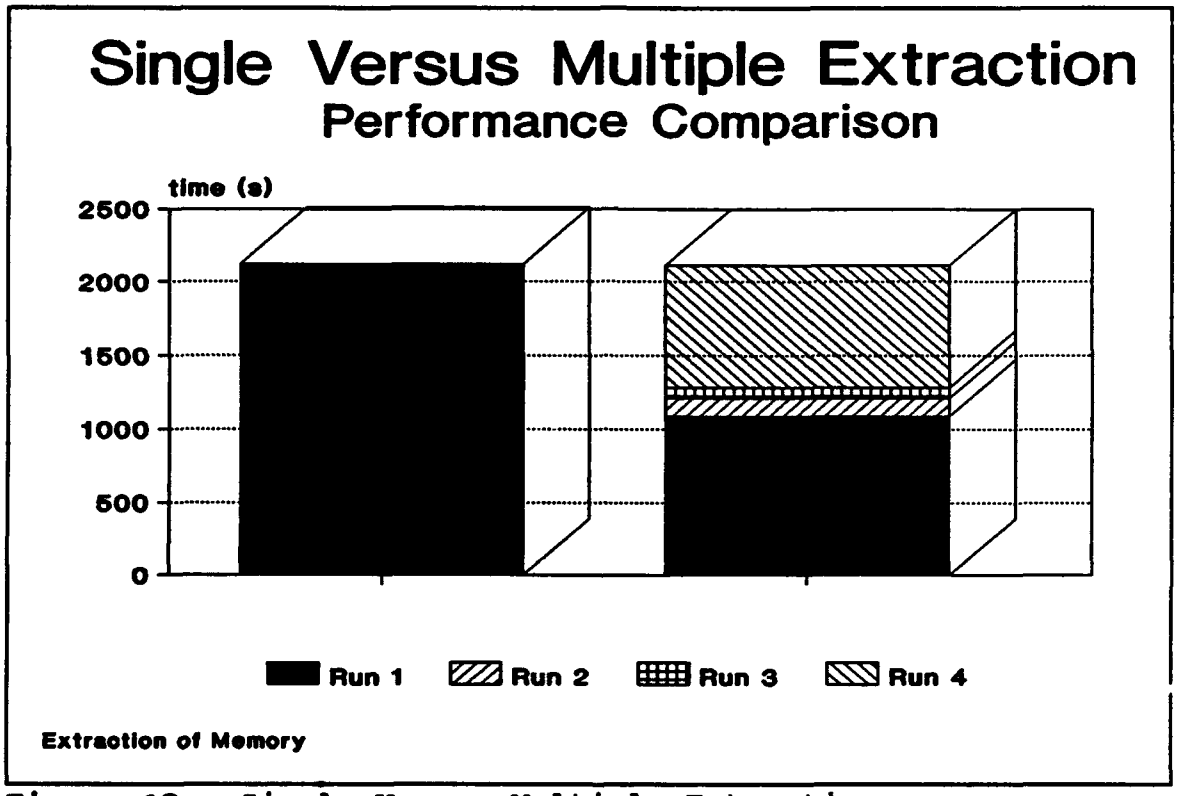


Figure 17. Single Versus Multiple Extractions.

After the optimized GENTEXN was shown to be both correct and faster, it was needed to show that the single versus multiple extraction comparison was still valid. Figure 17 shows such a comparison from the modified GENTEXN. The circuit is the memory circuit discussed previously. Figure 17 shows that extracting the circuit in one run or four runs yield approximately the same extraction times. In fact, with extraction times of over thirty-seven minutes,

the difference was approximately one second in favor of the multiple extraction. This difference is well within the one percent tolerance of the measurements.

6.4 Template Reduction

During the course of this thesis, it was noticed that performance was increased by using smaller templates and by fully exploiting the ability to hierarchically extract components. The following algorithm for optimally reducing the size of the templates has not yet been fully examined or tested. It is presented here as an idea for future work.

This algorithm is based on reducing the number and complexity of templates. This particular idea, however, may not increase performance on all circuits due to the high overhead. However, the potential increase in circuits with large templates cannot be ignored. This idea deals with decreasing the $(N!)^M$ permutability problem discussed in Section 3.5. This method can reduce the problem down to (N! * M) or better without any information being lost or ignored through a method called template reduction. This method can be applied with no additional information being required from the user. In fact, this method can be used with the existing GENTEXN user interfacing formats so that it will be transparent to the user.

As mentioned earlier, this method involves template reduction. The general idea of the method is to break up

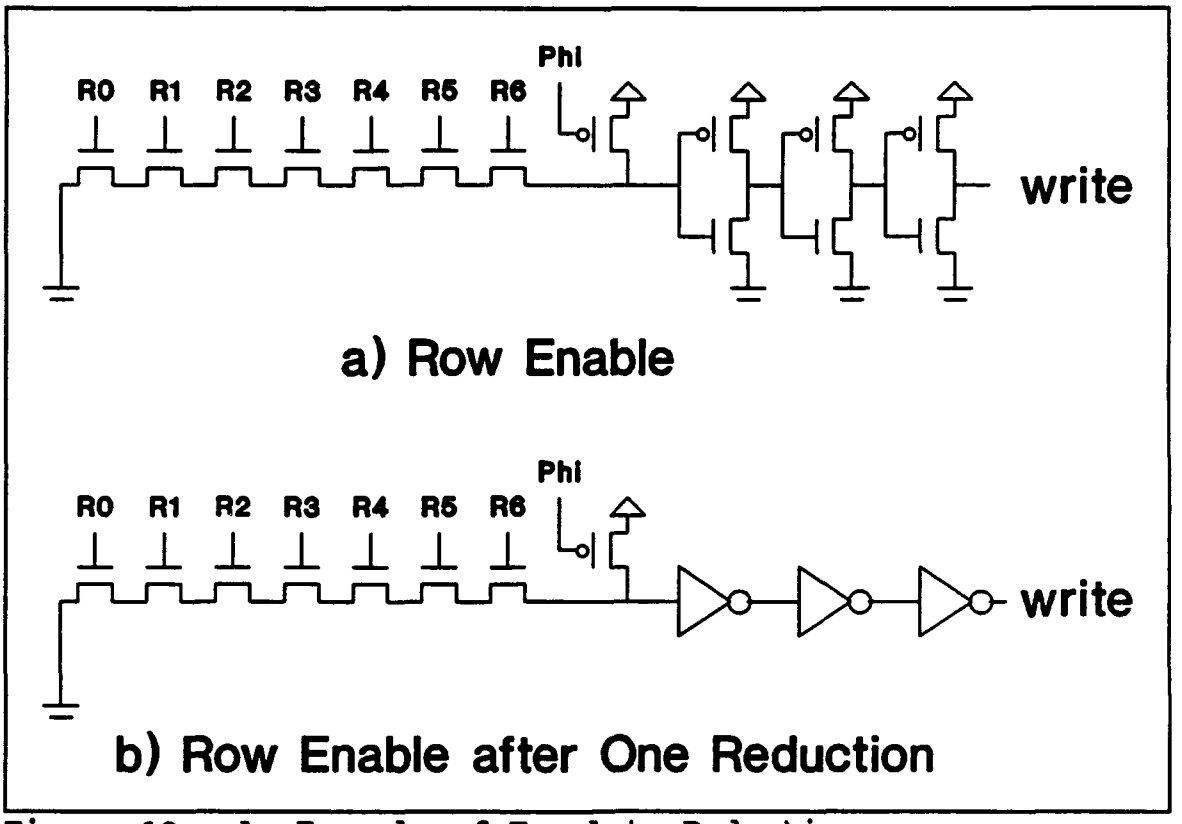


Figure 18. An Example of Template Reduction.

large templates into small temporary pieces. Figure 18 shows an example of this. Figure 18a has a total of 14 transistors. If the template is written for the row enable in terms of transistors only, there are 2^{14} or 16384 total permutations of the template. GENTEX treats each permutation as a separate template so it would actually extract 16384 templates, each with 14 transistors, from the circuit. However, if template reduction is applied to this template, the number of templates that GENTEX extracts is 20, each with four or less components. So, the number of templates is reduced by a factor of approximately 819. The complexity of the templates is reduced by a factor of 7,

which increases performance by at least that amount, for a total potential performance increase of a factor of 5733. Unfortunately, the actual increase in performance will be much less than that due to the high overhead of the method.

This is how it works. Subcomponents of the template are grouped into sub-templates. The size of the sub-template depends the number of subcomponents of the original template with and without permutable pins. No more than two subcomponents with permutable pins should ever grouped into a sub-template. If a sub-template contains one or more subcomponents that do not have permutable pins, then the sub-template should include exactly one subcomponent with permutable pins if one remains. There is no limit on the number of subcomponents in a sub-template that do not have permutable pins. Finally, all subcomponents within a sub-template must be interconnected.

Applying these rules to the schematic in Figure 18a shows how the factor of 5733 was obtained. Since all of the components in Figure 18a are transistors, the first sub-template should be two transistors. In this case, the inverters at the end of the circuit were chosen yielding Figure 18b. The template for the inverter had four permutations. The next sub-template should consist of all three inverters plus the p-type transistor. This sub-template had two permutations. The temporary component

produced by this sub-template should then be combined with the n-type transistor for the next sub-template. This process continues until the entire row enable circuit is extracted. The number of permutations that GENTEX must extract is four for the inverters plus two for each of the remaining eight transistors for a total of twenty.

There are quite a few problems associated with template reduction. The first problem is the overhead associated with generating the sub-templates. The performance gain also decreases rapidly for smaller templates. Finally, temporary components made during extraction must be restored to their original components if they are not used. This approach definitely needs further analysis.

6.5 Summary

This chapter discussed several changes and improvements made to the extraction programs. Memory usage was reduced by increasing memory management and by decreasing the size of the data structures. Performance was greatly increased by removing an unnecessary linked list, by fine tuning routines, and by specializing the duplicate search routine. These modifications were not only tested to work properly, but also tested to ensure they did not affect the comparison of single versus multiple extractions. The newly modified version of GENTEXN was shown to have excellent performance compared to other extraction tools. Finally, a new

algorithm, called template reduction, to optimally reduce the size and complexity of templates was presented and discussed.

VII. Results of EDIF Interfacing

7.1 Introduction

This chapter presents the results of the attempt to interface the extraction program, GENTEX, with the schematic tools at AFIT. The problems encountered are discussed, the limitations on the programs are introduced, and examples of the translations are given.

7.2 Problems Encountered

There were many problems encountered with the EDIF interfacing. Most of the problems were due to apparent errors in other software packages.

EDIF to Template. There were four main problems encountered with the EDIF2TMP utility, one problem for each way to generate EDIF plus one problem that occurred in all three. The EDIF outputs produced by Synopsys, CAPFAST, and GEN2EDIF are all slightly different and have their own unique problems.

SGE of Synopsys does not support the translation of schematics directly into EDIF. In order to generate EDIF in Synopsys, VHDL must be created, read into the design analyzer, optimized, and then written in EDIF. There are two problems with this. The process is lengthy and complicated. In addition, the circuit produced after this process, is in all likelihood, not the same as the original schematic. This makes it nearly impossible to generate

specific templates to test.

CAPFAST provides the best EDIF of the three sources. The EDIF produced through CAPFAST exactly matches the schematic. The EDIF also is the only one of the three sources that contained permutability information. There are two small problems with producing EDIF from CAPFAST. First, the EDIF differs from the EDIF produced by Synopsys. This required several modifications to EDIF2TMP. Second, generating the EDIF is not straight-forward. New symbols must be given the property "primitive:unknown", and if standard cell components from one of the CAPFAST libraries are used, a "-Q lib_name" option may or may not be required when producing the EDIF. The "lib_name" refers to the library for which the cell is marked as primitive, and is determined by examining the properties of the symbol.

GEN2EDIF produces EDIF easier than the other two sources, but the EDIF provides none of the inputs or outputs of the overall circuit. This information cannot be provided since the inputs and outputs of the overall circuit cannot be determined from the net list alone. The user must add the system inputs and outputs manually into the schematic.

The final problem encountered with the EDIF to template translation concerned the order of the pins of the components. GENTEX requires that the order of the pins of a particular component be fixed. When EDIF is produced by either CAPFAST or Synopsys, the order of the pins is not

necessarily the same as previously defined. The simplest way to avoid GENTEX from searching for an incorrect template due to this problem is to manually correct the order of the pins in the EDIF file. The order of the pins only need to be correct in their first occurrence in the cell definition within the EDIF.

GENTEX to EDIF. There were three problems encountered when converting a netlist to EDIF using GEN2EDIF. The first problem was that no overall circuit inputs and outputs could be determined from the netlist and therefore the schematic representation of the EDIF generated also lacked inputs and outputs. The second problem was that the EDIF produced did not translate smoothly into the SGE format. The pins of the gates and the wires connecting them, did not line up correctly. The wires would connect some of the pins and miss others. No pattern could be determined. The EDIF was thoroughly checked by hand and was checked in CAPFAST. No error could be found in the EDIF when checked by hand, and CAPFAST translated the EDIF correctly in all cases. Therefore, the problem was determined to be with SGE. The third problem was that routing the wires between connected pins of components was a complex problem. GEN2EDIF does not contain a complex routing routine. This leads to inefficient wiring and sometimes the inability to connect pins in a proper manner. If the routing routine cannot route a wire, the wire is connected directly and a warning

message is printed. This causes wires to be routed at all angles in the schematic instead of the 45° increments required by EDIF. CAPFAST handles the improper wires quite well and did some simple routing of the wires on its own. SGE, on the other hand, handles the improper wires very poorly. A portion of the improper wire may or may not show up in the schematic. Sometimes unrelated wires may not show up in the schematic for some unknown reason. Overall, the schematics generated by SGE were poor.

7.3 EDIF-to-Template Translation

The translation of EDIF to the template format used by GENTEX using EDIF2TMP produced expected results. Most of the problems encountered were expected or explainable.

CAPFAST provided the only means available to create a schematic and to directly convert it to EDIF, and it was

```
begin template;
    dff Vdin Vphi1 Vphi1_bar Vphi2 Vphi2_bar Vdout;
    t-gate Vdin:1 Vphi1_bar Vphi1 Vn1:1;
    inverter Vn1 Vn2;
    clk_inv Vn2 Vphi1 Vphi1_bar Vn1;
    t-gate Vn2:1 Vphi2_bar Vphi2 Vn3:1;
    inverter Vn3 Vdout;
    clk_inv Vdout Vphi2 Vphi2_bar Vn3;
end template;
```

Figure 19. Sample Template Produced by EDIF2TMP.

therefore used most to test the program. Figure 19 shows the template of a D-flip-flop produced from the EDIF of CAPFAST. The template has all the permutations correctly

marked as well as correctly showing the interconnectivity of the components.

SGE does not allow for the direct conversion of schematics to EDIF. Synopsys 3.1 is scheduled to be released in December of this year and has been reported to correct this problem. EDIF is generated through the design analyzer with VHDL. The schematics produced by this method did not in any case match the schematic used to produce the VHDL. The design analyzer has a library of components that it recognizes. The EDIF produced by the design analyzer contains buses of signals instead of listing the signals individually. EDIF2TMP does not support arrays and "RIPPER" cells (special cells used to connect individual signals to arrays of signals). The use of buses has to be turned off via the "-no_bus" option when using the "create_schematic" command in the design compiler. The resultant EDIF produces a correct template for the schematic that is produced. However, no permutability information is included in the EDIF and therefore is not in the template.

GEN2EDIF produced EDIF from a netlist that was compatible with EDIF2TMP. This was useful in the instances where it was desired to generate a template from one of the subcells in the Magic layout of a circuit. A ".sim" file was created of the sub-cell and extracted with the existing templates. The resultant netlist was then translated into a schematic in the EDIF format. This EDIF was translated into

a template. The pins of the new component had to be entered manually however. This was due to the fact that the EDIF provided by GEN2EDIF does not include the necessary input/output information. This approach was very useful in the extraction of custom and unknown circuits.

7.4 GENTEX-to-EDIF Translation

GEN2EDIF produces EDIF from netlists that can be successfully read by CAPFAST, SGE, and EDIF2TMP. The EDIF produced by GEN2EDIF has some problems as mentioned above. No inputs or outputs of the overall circuit are included and the wires cannot always be routed. This made the schematics hard to read. The wiring of the components can be turned off with a command line option while the node names are still printed on each pin. This allows the user to manually wire a circuit too confusing to read otherwise, or in which errors are occurring in the translation from EDIF to the schematic. The components are located in the schematic in the same relative position as in the transistor layout of the circuit. However, the component spacing is adjusted to fill the sheet size in SGE to reduce the overlapping of symbols. This does not necessarily provide for an easily readable schematic. Figures 20 and 21 show a schematic in CAPFAST and its close-up view, respectively, that was generated from the EDIF of GEN2EDIF with the wiring on. Once the schematic is read into either CAPFAST or SGE, the components can be rearranged into a more recognizable

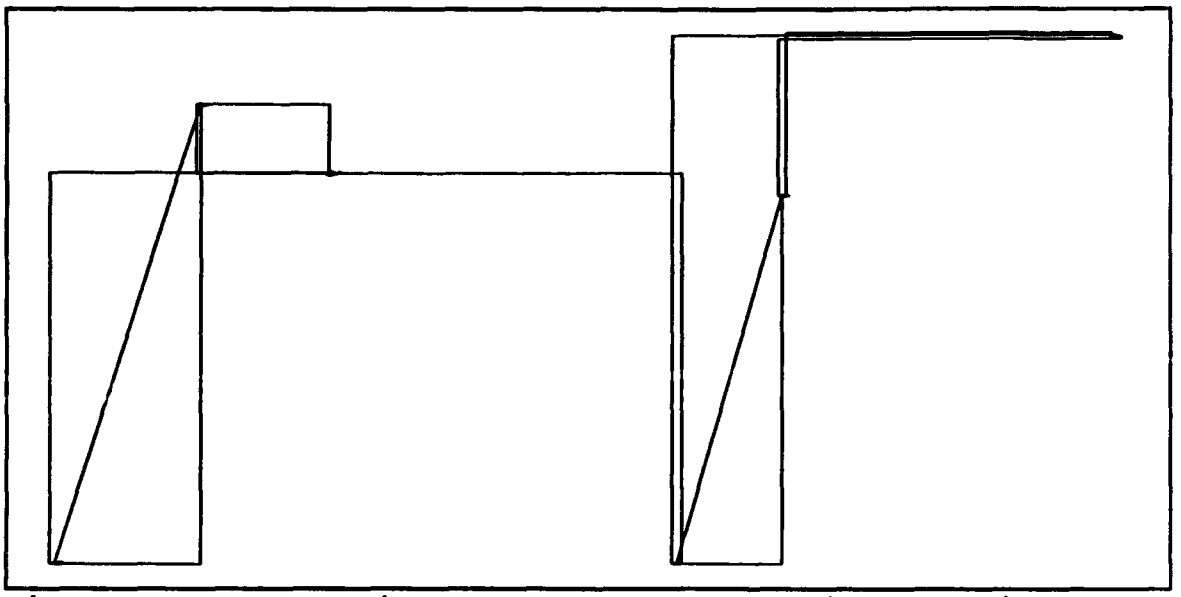


Figure 20. Schematic of a Master/Slave Flip-Flop with Clear from GEN2EDIF.

configuration. CAPFAST is the better tool for this purpose.

7.5 Summary

The translation to and from EDIF had limited success.

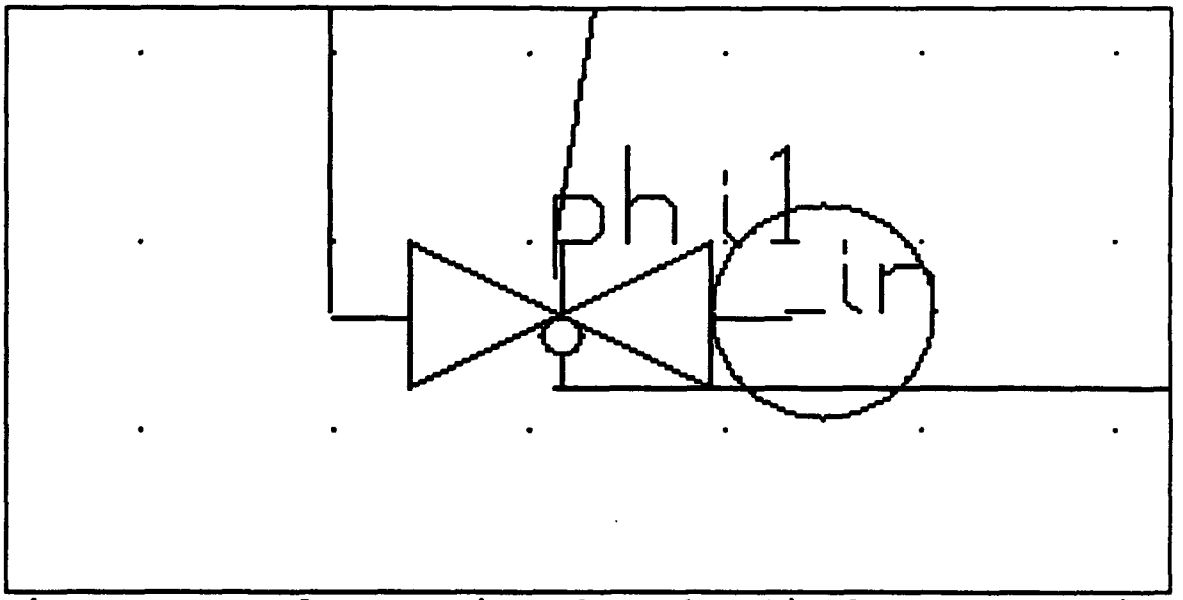


Figure 21. A Close-up View of a Schematic from GEN2EDIF in CAPFAST.

EDIF, although a national standard, is not completely compatible between software packages. EDIF2TMP sometimes generates templates missing permutability information due to the information not being present in the EDIF. GEN2EDIF cannot determine the inputs or outputs of the overall system from the netlist. GEN2EDIF cannot always route a wire which creates problems in translating the EDIF to a schematic. Finally, SGE appears to have many problems reading EDIF, that according to other tests has no errors.

VIII. Conclusions and Recommendations

8.1 Conclusions

There are three important conclusions that can be determined from the results of this thesis. These conclusions are discussed and justified in Chapters 5, 6, and 7.

The first conclusion is that the algorithm used in GENTEXN both outperforms and is less affected by changes in the type of circuit or template than the other five algorithms. GENTEXN is the algorithm that searches for potential matches to the templates by node and then checks the number of pins connected to the node before accepting the match. The overhead involved with such a search appears to be proportional with the size of the circuit. The overhead involved with GENTEXC and GENTEXCN appears to increase exponentially with the size of the circuit. The overhead involved with GENTEXO and GENTEXON appears to increase linearly with the size of the circuit, but the number of output nodes went up linearly also. The number of output nodes increased to the point that the advantage of knowing the output of the gate was lost. It became faster to start the search from Vdd or GND as GENTEX and GENTEXN generally did. Furthermore, after GENTEXN was optimized in Chapter 6, it had excellent performance compared to other extraction tools, including GES. The simplicity of the

templates of the GENTEX programs is also a significant advantage over other extraction tools.

The second conclusion from the results of this thesis is that the templates may affect the performance of the extraction more than the extraction algorithm. It was found that templates should generally be ordered from the one that has the greatest number of matches to the one that has the least number of matches. The purpose of ordering the templates in this manner was to decrease the number of components in the circuit as much as possible for each subsequent search. The order of the subcomponents within the template also plays an important role in performance. Writing efficient templates is difficult but is separated into three simple rules: 1) literal names first, 2) list subcomponents in order of connection, and 3) least common subcomponent first. These three rules have a single common goal. The goal of the three rules are to make any potential match of an incorrect circuit fail as quickly as possible.

The third and final conclusion of this thesis concerns EDIF interfacing. EDIF was useful in the verification process. However, the system is far from being able to be automated. The present EDIF utilities in CAPFAST and SGE are not straightforward and require much more time to use than should be necessary. The problem with GEN2EDIF not being able to determine the inputs and outputs of the circuit is not a major concern. The user generally knows

with just a glance where they should go. The node names provide many clues to the inputs and outputs for more difficult circuits.

8.2 Recommendations

These recommendations consist of both potential solutions to some of the problems encountered and some new ideas that may work better than the methods used in this thesis. The results of these recommendations are in most cases unknown. Future work is required to determine the effectiveness of these recommendations.

Decreasing the Amount of Memory Used. Although the amount of memory used was not a problem for the circuits used in this thesis, it may be a problem in the future. The largest circuit tested (140,000 transistors) combined with the templates used to extract it used approximately 29M (29 Megabytes) of memory. The Sun SPARC II computers that were used had either 16M, 32M or 64M of memory. This circuit could not be run on the computers with 16M of memory without paying a stiff performance penalty due to disk swapping. The circuit used was also close to the 32M limit of many of the other computers. In order to decrease the amount of memory used by GENTEX, several things can be done.

There exist many inefficiencies in the code. The memory of the subcomponents consumed in the process of creating a new component is not reused. It is simply discarded. A modification made to GENTEX to incorporate a

fix to this problem is discussed in Chapter 6. The memory usage went down to 26M with no performance lost. In some circuits, the performance actually got better due to the decreased need to ask the operating system for more memory. There is still much work that can be done in this area.

The data structures are also memory inefficient. In many cases, such as the node data structure, there is a text field for the name. This field is a fixed length and therefore was purposely made to be large (100 characters) so that it would be big enough for the biggest name expected to be encountered. These fields can be changed to pointers instead of arrays of characters. When a name is read from the data file, memory just big enough to hold the name may be allocated and the pointer in the data structure can then be changed to point to it. This is expected to have a significant impact on the amount of memory used without a significant decrease in performance.

Finally, the templates tend to use a lot of memory. The biggest memory problem with the templates is with the pins of the subcomponents. The change in the data structures discussed in the previous paragraph should alleviate a good portion of this problem. Another potential fix is to read only one template at a time into memory. The templates could be read as they were needed, reusing the memory. The effect of this change on performance is unknown, and the significance of the memory usage decrease

depends on the number and size of the templates used in any particular run.

Increasing Performance. The performance of GENTEXN was the best of the six algorithms tested in this thesis, but it could be better. Aside from finding a faster algorithm, increasing the efficiency of the code is the only known way of increasing the performance of the program. There are some modifications discussed in Chapter 6 that dramatically increased performance. Other code optimizations are possible, but it is unlikely that significant increases in performance can be obtained in a single processor environment. Due to the highly parallel nature of GENTEX, the next logical step would be to re-code the program in a parallel processing language.

The only other way to increase performance is to find a better algorithm. The template reduction method discussed in Chapter 6 is one possibility. Some of the alternative solutions discussed in Chapter 3 should also be considered.

Increasing User-Friendliness. GENTEXN is not interactive; so, being user-friendly is not as big a problem as it could be. However, this program does lack somewhat in notifying the user of the errors that have occurred. Most errors in the templates or ".sim" file are caught and the user is notified, but some errors produce either generic messages or may even still cause a program crash. There is a lot of time and effort that could be spent in this area,

depending on the level of error checking and notification desired.

One idea that would be relatively simple to implement but could save the user a tremendous amount of work is to allow arrays of pins in the templates. For large components, like memory blocks, the number of pins makes it impractical to write many templates that include that component. The 1K RAM cells extracted in this thesis required 640 pins to be declared. The task of writing the template would have been much faster with the use of several arrays of pins rather than declaring each pin individually.

Appendix A: User's Manual

A.1 Introduction

This appendix contains the user's manual for the verification system created in this thesis. The two EDIF translation routines, EDIF2TMP and GEN2EDIF, as well as GENTEX are described in terms of their operation and use. Finally, a checklist is given that can be used for the integrated use of all three programs.

A.2 EDIF2TMP

EDIF2TMP translates an EDIF schematic into a template for GENTEX. The schematic is assumed to be non-hierarchical, or flat. A hierarchy of templates may be generated using a hierarchy of schematics.

EDIF2TMP is the easiest to use of the three programs produced by this thesis. The usage command is "edif2tmp filename [-o outfilename]". The "filename" is assumed to have a ".edf" extension unless otherwise specified. The output will go to a file with the name of the input file with a ".tmp" extension unless otherwise directed with the "-o" option.

The one thing to beware of is in the EDIF. If the ports are declared in a different order in the EDIF than in other templates, EDIF2TMP will also produce the template with the ports in the wrong order. The first declaration of the ports in a cell may have to be manually rearranged into

the proper order before using EDIF2TMP.

A.3 GEN2EDIF

GEN2EDIF translates the component netlist output of GENTEX into an EDIF schematic. The usage statement is "gen2edif [-s schem_name] [-c] filename [-o outfilename]". The "filename" is assumed to have a ".out" extension (extension produced by GENTEX) unless otherwise specified. The output file name is the input filename with a ".edf" extension unless changed with the "-o outfilename" option. The top-level name of the schematic will be the name of the input file unless changed with the "-s schem_name" option. GEN2EDIF will connect and label the names of the nodes connecting the pins of the components. However, sometimes this makes the schematic hard to read, and sometimes GEN2EDIF is unable to route a wire properly. In these cases, the no connection, "-c", option to turn off the wiring may help.

GEN2EDIF needs several files in order to work properly. A library file is needed for the same reasons that GENTEX needs one. The format and purpose of this file is discussed in the next section. The library file for GEN2EDIF is called "gen2edif.lib".

EDIF requires information such as if a pin is an input, output, or bi-directional. EDIF also requires a graphical representation of a component to be present. This information cannot be derived from the component netlist.

Therefore in order for GEN2EDIF to create EDIF schematics, another source for this information must be provided. GEN2EDIF looks for a file with the name of the component and a ".edf" extension. For example, GEN2EDIF looks in the file "inverter.edf" for the required information on an inverter. These ".edf" can be generated by translating a symbol to EDIF. SGE and CAPFAST have utilities to make this translation. SGE was mostly used for this purpose in this thesis. First draw the symbol in SGE, then convert it to EDIF. As with making schematics in EDIF, the ports may need to be rearranged in the cell declaration.

Another file used by GEN2EDIF is a configuration file. The present version of GEN2EDIF only recognizes one command in this file. The line "library=(path)" will tell GEN2EDIF what directory to look for the "component.edf" files in if it cannot find them in the default directory. This allows the user to put all the "component.edf" files into a single directory or to share them with another user.

The final file used by GEN2EDIF is a file called "lib.edf". This file contains header information for the EDIF that is generated. Nothing needs to be done with this file; however, it must be present in either the default directory or in the library directory.

A.4 GENTEX

The GENeric Template EXtractor, GENTEX, is the nucleus of the verification system. GENTEX actually does the work.

EDIF2TMP and GEN2EDIF are programs used to support it. GENTEX hierarchically extracts a component netlist (usually a transistor netlist) based on the templates provided by the user.

Functions. GENTEX has three distinct functions it performs on the component net list. Any or all of these functions may be chosen with command line options. The input to GENTEX is generally either a ".sim" file produced by either MEXTRA or EXT2SIM, or it can be the output file of a previous run of GENTEX.

The first function is to eliminate any duplicate transistors. Any two transistors that have all of their respective pins connected to the same nodes are said to be duplicates. Generally duplicate transistors are used in place of a single transistor where the available space in the layout does not allow for a single transistor of the desired size. These duplicates may be combined into a single transistor with the new width being equal to the sum of the widths of the two transistors combined. GENTEX combines all duplicate components in this manner.

The second function of GENTEX is to extract inverters and transmission gates (t-gates). These two extractions are separated from the other extractions because these two components are the most common in CMOS and therefore need to be extracted in almost every circuit. In order to speed up the extraction of these two components, the extraction

routines for them are hard-coded.

The third function of GENTEX is to extract the circuit into the new components defined in the templates. This function is the heart of GENTEX. The components in the circuit used to make a match to a template are deleted from the circuit and replaced with the new component defined in the template. The circuit may be hierarchically extracted in this manner to the desired level of abstraction.

Command Line Options. GENTEX has the following usage statement: "gentex [-itbdT] simfilename [-o outfilename]". The "simfilename" is the input file name. A ".sim" extension is assumed unless otherwise stated. The output file name is normally the input file name with a ".out" extension instead of the ".sim" extension. The name of the output file can be changed with the "-o" option. The new name of the output file becomes the "outfilename" specified in the command line. No extension is added to "outfilename".

The other five command line options tell GENTEX to skip a process that is normally performed. Multiple options may be listed separately or together. For example, "gentex -iT filename" is the same as "gentex -i -T filename." The "-i" and "-t" options tell GENTEX to skip the extraction of inverters and t-gates, respectively. If the extraction of both gates needs to skipped, either "-it" or "-b" may be used. The "-d" option tells GENTEX not to look for and

remove duplicate transistors. Finally, the "-T" tells GENTEX not to extract the components defined by the templates.

Library. The library is an important part of GENTEX. When GENTEX reads a component netlist, it needs to know the number of pins of each of the different component types. The library is one way of giving this information to GENTEX. The library file, "gentex.lib," contains a list of components followed by their number of pins. For example, "NAND 3" could be one line in the file. New components that are defined in a template do not need to be listed in the library file. The number of pins can be determined from the information in the template. Inverters, t-gates, p-type transistors, and n-type (or e-type) transistors also do not need to be included in the library file.

If a component name is read that is not recognized by GENTEX, it is assumed to be a comment and ignored. So it is important to put the names of any components unknown to GENTEX that are in the input file into the library file.

Templates. The templates are the most important part of GENTEX. The templates define new components in terms of existing components and are located in the file "gentex.tmp". New components may even be defined in terms of components defined in previous templates. Figure 22 shows an example template file.

There are several things that need to be discussed

```
begin template;
    nand Vin1 Vin2 Vout;
    p Vin1 LVdd:1 Vout:1;
    p Vin2 LVdd:1 Vout:1;
    n Vin1 Vout:1 Vn1:1;
    n Vin2 Vn1:1 LGND:1;
end template;

begin template;
    and Vin1 Vin2 Vout;
    nand Vin1:1 Vin2:1 Vn1;
    inverter Vn1 Vout;
end template;
```

Figure 22. Sample Template File.

regarding the sample templates in Figure 22. Notice that each template is bounded by the lines "begin template;" and "end template;". The first line in each template defines the new component and its pins. The first template defines a NAND gate and the second defines an AND gate. Notice that the NAND gate defined in the first template is used in the definition of the second template.

The most important parts of the template are the node names. The names after each component name in the template represent the names of the nodes to which each of the pins of the component are connected. Pins that are connected together have the same node name. The pins of each component must be reference in the same order each time. For example, the NAND gate was defined in the first template such that the first pin was its first input and the last pin was the output. When the NAND gate is used in the second template, the order of the pins must be the same as the

order in which they were defined. Inverters, t-gates, p-type transistors, and n-type transistors already have the order of their pins defined (see Figure 23). Other components have the order of their pins defined by their template.

e	gate drain source
p	gate drain source
inverter	input output
t-gate	input (gate of 'p') (gate of 'e') output

Figure 23. Pin Order for Existing Components in GENTEX.

Notice that all of the node names in the template begin with either "V" or "L". The "V" stands for a variable node name, and the "L" stands for literal node names. In most circumstances, the literal node names are only used to define a connection to "Vdd" or "GND", but they may be used to define a connection to any specific node. The variable node names are used whenever the node does not have to be a specific node. These names eventually get mapped to a specific node during the extraction process.

The last thing to note about the templates are the ":1"s that are attached to some of the node names. The ":1" is not part of the node name. It is used to tell GENTEX that pins of a component are interchangeable, or permutable. Notice that the drain and source of the transistors as well as the two inputs to the NAND gate are marked as permutable.

If a three-input NAND gate were used, all three of the inputs would be marked with a ":1". The number after the colon does not need to be a one. It can be any positive integer below 32768. Any pins marked with the same number are interchangeable within that component. In fact, different sets of pins may be marked as permutable with each other within the same component by using different numbers. Notice that the pins of the new component in each template are not and should not be marked as permutable.

A.5 A Checklist

The following is a checklist of how most verification processes would be performed. The list may be modified to fit the individual situations.

1. Make an EDIF file of each of the schematics of the components used in the design. Currently, this must be done with CAPFAST, but Synopsys 3.1 scheduled to be released in December of this year should also be able to generate the EDIF files.

2. Make sure that the port order in the cell declarations of the EDIF is consistent with the desired order. The ports will be listed in the templates in the order listed in the cell declaration of the EDIF. So, it is important to make sure the order is correct.

3. Make templates of each component using EDIF2TMP. Each template will be generated based on the information in the EDIF file.

4. Write any necessary templates by hand. If there are any templates that are needed and no schematic exists for them, they must be written by hand.

5. Combine the templates into a single file. Make sure that the templates of components that are defined in terms of other components are after the templates of those components.

6. Run GENTEX on the ".sim" file. Make sure that the templates are located in the file "gentex.tmp". If the extraction is done in more than one run, make sure that the proper component names and the number of their ports are in "gentex.lib".

7. Generate the EDIF of any new symbols used in the schematic. The EDIF should be placed in either the default directory or added to the directory defined as the library.

8. Check the EDIF of the new symbols to make sure the ports are in the correct order. The order of the ports in the cell declaration of the EDIF file must be in the correct order.

9. Translate the output of the extraction into a schematic in EDIF. Use GEN2EDIF on the output file of the extraction process.

10. Read the schematic into a schematic capture tool. The new schematic is the actual layout and interconnection of higher-level components in the circuit layout.

11. Verify the circuit. If the circuit was extracted

to a high enough level of abstraction, this may be done visually. Otherwise, a simulator is required.

A.6 Summary

A brief description and tutorial was given on the use of the three programs generated by this thesis. Some of the problems of which to beware were brought up, and ways around these problems were presented. Finally, a checklist was provided that demonstrates the typical verification process using the tools provided.

Appendix B: Program Code

Due to the size of the program code, it has been included into a separate volume. A copy of the code may be obtained through:

Major Kim Kanzaki, USA
Air Force Institute of Technology
AFIT/ENG
Wright-Patterson AFB, OH 45433-7765
(513) 255-3576

Bibliography

1. HSPICE User's Manual. Meta-Software Inc. Campbell, CA, 1991.

2. Dukes, Michael A. Hardware-Verification Through Logic Extraction. PhD dissertation. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1992 (AAI-8835).

3. Camurati, Paolo and Paolo Prinetto. "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," Computer, 21: 8-19 (July 1988).

4. Terman, Chris. "ESIM," Berkeley Distribution of Design Tools. Computer Science Division, EECS Department, University of California at Berkeley, 1986.

5. Barrow, Harry G. "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," Artificial Intelligence, 24: 437-491 (1984).

6. Boyer, Robert S. A Computational Logic. New York: Academic Press, 1979.

7. Petre, Mark. "STOVE: Sim TO VHDL Extraction," AFIT VLSI CAD Tools. Systems Research Laboratories, 1988.

8. Gallagher, David M. Rapid Prototyping of Application Specific Processors. MS thesis, AFIT/GE/ENG/87D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A189541).

9. Dukes, Michael A. A Multiple-Valued Logic System for Circuit Extraction to VHDL 1076-1987. MS thesis, AFIT/GE/ENG/88S-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, September 1988 (AD-A202646).

10. Papaspyridis, Alexander C. "A Prolog-Based Connectivity Verification Tool," Proceedings of the IEEE International Conference on Computer-Aided Design. 523-527. New York: IEEE Press, 1988.

11. De Loore, B. J. S. and A. P. Kostelijk. "Automatic Verification of Library-Based IC Designs," Proceedings of the IEEE Custom Integrated Circuits Conference. 30.6.1-30.6.5. New York: IEEE Press, 1990.

12. Spickelmier, Rick L. and A. Richard Newton. "Connectivity Verification Using a Rule-Based Approach,"Proceedings of the IEEE International Conference on Computer-Aided Design. 190-192. New York: IEEE Press, 1985.

13. Yarost, Stuart A. A Circuit Extraction System and Graphical Display For VLSI Design. MS thesis, AFIT/GCE/ENG/89D-9. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215668).

14. Nebel, Wolfgang and Reiner W. Hartenstein. "Functional Design Verification by Register Transfer Net Extraction from Integrated Circuit Layout Data," COMPEURO. 254-257. New York: IEEE Press, 1987.

15. Boehner, Michael. "LOGEX - An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology," Proceedings of the IEEE/ACM Design Automation Conference. 517-522. New York: IEEE Press, 1988.

16. Martin, Harold and Ranjani Ram. "A CAD Tool for Circuit Diagram Extraction from VLSI Layout Cells," Midwest Symposium on Circuits and Systems. 817-820. New York: IEEE Press, 1989.

Vita

First Lieutenant Kenneth J. McClellan Jr. was born 14 December 1966 in Mariemont, Ohio. He graduated from Loveland Hurst High School in Loveland, Ohio in 1985 and attended the U.S. Air Force Academy, graduating with a Bachelor of Science in Electrical Engineering (specialty: Digital Design) and a second major of applied physics in May 1989. Upon graduation, he received a regular commission in the USAF and served his first tour of duty at Wright-Patterson AFB, Ohio. He began as a project officer for the Productivity, Reliability, Availability, and Maintainability (PRAM) Program Office where he managed several multi-million dollar projects until June 1990. He was then assigned to work as a project engineer for the Reliability And Maintainability Technology Insertion Program (RAMTIP) Office and the PRAM Program Office. There he was responsible for evaluating potential projects as well as overseeing the technical aspects of ongoing projects until entering the School of Engineering, Air Force Institute of Technology, in June 1991.

Permanent Address: 6504 Branch Hill-Miamiville Rd.
Loveland, Ohio 45140

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
A GENERIC TEMPLATE EXTRACTOR (GENTEX) IN C FOR VLSI DESIGN VERIFICATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Kenneth J. McClellan, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/92D-25

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/ENG
WPAFB OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problem of VLSI design verification through circuit extraction was analyzed. The problems of creating a simple template format, the permutability of pins, maintaining connectivity, and performance were focused on. A generic template extractor (GENTEX) was developed in the C programming language for use as a testbed to find solutions to these problems. Six different extraction algorithms were tested with GENTEX and compared based on performance. EDIF translation programs were used to interface with GENTEX on both the input and output sides. One translation program converted an EDIF representation of a schematic into the template format used by GENTEX. The other translation program converted the output of GENTEX into a schematic in EDIF. The results of the performance analysis showed that an extraction algorithm based on searching the data structures by node rather than by component type provided the best performance. The results also showed that comparing the number of connections to a node within a template to the actual number of connections to a node within the circuit being extracted, not only eliminated any connectivity problems but also increased performance.

14. SUBJECT TERMS
VLSI, VLSI Verification, Circuit Extraction, CAD Tools

15. NUMBER OF PAGES
123

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL